AD-A077 522　PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING　F/G 5/8
PASM: A PARTITIONABLE MULTIMICROCOMPUTER SIMD/MIMD SYSTEM FOR I--ETC(U)
AUG 79　H J SIEGEL , L J SIEGEL , F KEMMERER　F30602-78-C-0025
UNCLASSIFIED　TR-EE79-40　AFOSR-TR-79-1156　NL

1 OF 1
AD-
A077522

END
DATE
FILMED
1 - 80
DDC

PASM: A PARTITIONABLE MULTIMICROCOMPUTER SIMD/MIMD SYSTEM FOR IMAGE PROCESSING AND PATTERN RECOGNITION

Howard Jay Siegel
Leah J. Siegel
Frederick Kemmerer
Philip T. Mueller, Jr.
Harold E. Smalley, Jr.
S. Diane Smith

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

TR-EE 79-40
August 1979

This work was principally supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR-78-3581. It was supported in part by the U.S. Air Force, Rome Air Development Center, under Contract No. F30602-78-C-0025.

(4)

(9) Interim rept. Mar 78-Aug 79,

(6)

PASM: A PARTITIONABLE MULTIMICROCOMPUTER SIMD/MIMD SYSTEM FOR IMAGE PROCESSING AND PATTERN RECOGNITION.

(10)
Howard Jay/Siegel,
Leah J./Siegel,
Frederick/Kemmerer,
Philip T./Mueller, Jr.
Harold E./Smalley, Jr/
S. Diane Smith

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

(18) AFOSR

(19) TR-79-1156

Purdue University

(14) TR-EE 79-40

(11) Aug[redacted]979 (12) 70

This work was principally supported by the Air Force Office of Scientific Research, Air Force Systems Commands, USAF, under Grant No. AFOSR-78-3581. It was supported in part by the U. S. Air Force, Rome Air Development Center, under Contract No. F30602-78-C-0025, ✓AFOSR-78-3581 (15)

79 11 23 026 ✓

292 000 mt

A

PASM: A PARTITIONABLE MULTIMICROCOMPUTER SIMD/MIMD SYSTEM FOR IMAGE PROCESSING AND PATTERN RECOGNITION

Howard Jay Siegel
Leah J. Siegel
Frederick Kemmerer
Philip T. Mueller, Jr.
Harold E. Smalley, Jr.
S. Diane Smith

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907
August 1979

ABSTRACT

PASM, a large-scale multimicroprocessor system being designed at Purdue University for image processing and pattern recognition, is described. This system can be dynamically reconfigured to operate as one or more independent SIMD and/or MIMD machines. PASM consists of a Parallel Computation Unit, which contains N processors, N memories, and an interconnection network; Q Micro Controllers, each of which controls N/Q processors; N/Q parallel secondary storage devices; a distributed Memory Management System; and a System Control Unit, to coordinate the other system components. Possible values for N and Q are 1024 and 16, respectively. The control schemes, interprocessor communications, and memory management in PASM are explored. Examples of how PASM can be used to perform image processing tasks are given.

This research was principally supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR-78-3581. It was supported in part by the U. S. Air Force, Rome Air Development Center, under contract No. F30602-78-C-0025.

B

PREFACE

Portions of this report have appeared in the following papers and reports:

Siegel, H. J., "Preliminary design of a versatile parallel image processing system," Third Biennial Conf. on Computing in Indiana, Apr. 1978, pp. 11-25.

Siegel, H. J., Kemmerer, F., and Washburn, M., "Parallel memory system for a partitionable SIMD/MIMD machine," 1979 Int'l. Conf. Parallel Processing, Aug. 1979, pp. 212-221.

Siegel, H. J., McMillen, R. J., Mueller, P. T., Jr., and Smith, S. D., A Versatile Parallel Image Processor: Some Hardware and Software Problems, School of Electrical Engineering, Purdue University, Technical Report TR-EE 78-43, Oct. 1978.

Siegel, H. J., Mueller, P. T., Jr., and Smalley, H. E., Jr., Preliminary Design Alternatives for a Versatile Parallel Image Processor, School of Electrical Engineering, Purdue University, Technical Report TR-EE 78-32, June 1978.

Siegel, H. J., Mueller, P. T., Jr., and Smalley, H. E., Jr., "Control of a partitionable multimicroprocessor system," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 9-17.

Siegel, H. J., Siegel, L. J., McMillen, R. J., Mueller, P. T., Jr., and Smith, S. D., "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition," 1979 IEEE Comp. Soc. Conf. Pattern Recog. and Image Processing, Aug. 1979, pp. 214-224.

TABLE OF CONTENTS

D

I. INTRODUCTION

As a result of the microprocessor revolution, it is now feasible to build multimicroprocessor systems capable of performing image processing tasks more rapidly than previously possible. There are many image processing tasks which can be performed on a parallel processing system, but are prohibitively expensive to perform on a conventional computer system due to the large amount of time required to do the tasks [34]. In addition, a multimicroprocessor system can use parallelism to perform the real time image processing required for such applications as robot (machine) vision, automatic guidance of air and space craft, and air traffic control.

There are several types of parallel processing systems. An SIMD (single instruction stream - multiple data stream) machine [12] typically consists of a set of N processors, N memories, an interconnection network, and a control unit (e.g. Illiac IV [7]). The control unit broadcasts instructions to the processors and all active ("turned on") processors execute the same instruction at the same time. Each processor executes instructions using data taken from a memory with which only it is associated. The interconnection network allows interprocessor communication. An MSIMD (multiple-SIMD) system is a parallel processing system which can be structured as two or more independent SIMD machines (e.g. MAP [25,26]). The Illiac IV was originally designed as an MSIMD system [3]. An MIMD (multiple instruction stream - multiple data stream) machine [12] typically consists of N processors and N memories, where each processor can follow an independent instruction stream (e.g. C.mmp [66]). As with SIMD architectures, there is a multiple data stream and an interconnection network. A partitionable SIMD/MIMD system [46] is a parallel processing system which can be structured as two or more independent SIMD and/or MIMD machines (e.g. PASM [38]).

PASM, a partitionable SIMD/MIMD machine, is a large-scale dynamically reconfigurable multimicroprocessor system being developed at Purdue University [38,43,44,46-49]. It is a special purpose system being designed to exploit the parallelism of image processing and pattern recognition tasks. It can also be applied to related areas such as speech processing and biomedical signal processing. In this paper, the architecture of PASM is presented and examples of its use in performing image processing tasks are given.

Due to the low cost of microprocessors, computer system designers have been considering various multimicrocomputer architectures, such as [6,8,18,21,22,29,60,63,64]. PASM was the first system in the literature to combine the following features:

(1) it can be partitioned to operate as many independent SIMD and/or MIMD machines of varying sizes, and

(2) a variety of problems in image processing and pattern recognition will be used to guide the design choices.

It was not until the current "microprocessor revolution" that designing a system with as many as 1024 full processors was feasible. Many designers have discussed the possibilities of building large-scale parallel processing systems, employing 2^{14} to 2^{16} microprocessors, in SIMD (e.g. binary n-cube array [29]) and MIMD (e.g. CHoPP [60,61]) configurations. Without the presence of such a large number of processors, the concept of partitioning the system into smaller machines which can operate as SIMD or MIMD machines was unnecessary. Nutt [25] has suggested a machine which is a multiple-SIMD system. Lipovski and Tripathi [22] have considered the idea of combining the SIMD and MIMD modes of operation in one system. In addition, developments in recent years have shown the importance of parallelism to image processing, using both cellular logic arrays (e.g. CLIP [56], BASE 8 [32]) and

SIMD systems (e.g. STARAN [33]). A variety of such systems are discussed in [13]. Thus, the time seems right to investigate how to construct a computer system such as PASM: a machine which can be dynamically reconfigured as one or more SIMD and/or MIMD machines, optimized for a variety of important image processing and pattern recognition tasks.

The use of parallel processing in image processing has been limited in the past due to cost constraints. Most systems used small numbers of processors (e.g. Illiac IV [7]), processors of limited capabilities (e.g. STARAN [33]), or specialized logic modules (e.g. PPM [19]). With the development of microprocessors and related technologies it is reasonable to consider parallel systems using a large number of complete processors.

SIMD machines can be used for "local" processing of segments of images in parallel. For example, the image can be segmented, and each processor assigned a segment. Then, following the same set of instructions, such tasks as line thinning, threshold dependent operations, and gap filling can be done in parallel for all segments of the image simultaneously. Also in SIMD mode, matrix arithmetic used for such tasks as statistical pattern recognition can be done efficiently.

MIMD machines can be used to perform different "global" pattern recognition tasks in parallel, using multiple copies of the image or one or more shared copies. For example, in cases where the goal is to locate two or more distinct objects in an image, each object can be assigned a processor or set of processors to search for it.

There are also tasks which require parallel processing in both SIMD and MIMD modes. As a simple example consider the task of determining if a line drawing contains a square. In SIMD mode a parallel processing system can segment the image and each processor can locally determine which points in its segment, if any, are possible corners of squares. The system can then switch to MIMD mode, where each corner will be assigned to a processor which examines the image globally to determine if the corner is actually part of a square. Another SIMD/MIMD application might involve using the same set of microprocessors for preprocessing an image in SIMD mode and then doing a pattern recognition task in MIMD mode.

Figure 1 is a block diagram of the basic components of PASM. The Parallel Computation Unit (PCU) contains N processors, N memory modules, and an interconnection network. The PCU processors are microprogrammable microprocessors that perform the actual SIMD and MIMD computations. The PCU memory modules are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The interconnection network provides a means of communication among the PCU processors and memory modules. Two possible ways to organize the PCU and different types of partitionable networks which can be used are described in section II.

The Micro Controllers (MCs) are a set of microprogrammable microprocessors which act as the control units for the PCU processors in SIMD mode and orchestrate the activities of the PCU processors in MIMD mode. There are Q MCs. Each MC controls N/Q PCU processors. A virtual SIMD machine of size RN/Q where $R = 2^r$ and $1 \leq r \leq q$, is obtained by loading R MC memory modules with the same instructions simultaneously. Similarly, a virtual MIMD machine of size RN/Q is obtained by combining the efforts of the PCU processors of R MCs. Possible values for N and Q are 1024 and 16, respectively. Control Storage contains the programs for the MCs. The MCs are discussed in more detail in section III.

The Memory Management System controls the loading and unloading of the PCU memory modules. It employs a set of cooperating dedicated microprocessors. The Memory Storage System stores these files. Multiple devices are used to allow parallel data transfers. The secondary memory system is described in section IV.

The System Control Unit is a conventional machine, such as a PDP-11, and is responsible for the overall coordination of the activities of the other components of PASM. Examples of the tasks the System Control Unit will perform include program development, job scheduling for all of the PCU, and coordination of the loading of the PCU memory modules from the Memory Storage System with the loading of the MC memory modules from Control Storage. By carefully choosing which tasks should be assigned to the System Control Unit and which should be assigned to other system components (such as the Memory Management System), the System Control Unit can work effectively and not become a bottleneck.

Together, sections II, III, and IV present the overall architecture of PASM. Particular attention is paid to the ways in which the control structure, interprocessor communications, and secondary memory scheme allow PASM to be efficiently partitioned into independent virtual machines. Variations in the design of PASM's PCU which still support these control, communications, and secondary memory ideas are examined. This examination demonstrates how the concepts underlying PASM can be used in the design of different systems.

In section V, image processing algorithms using PASM are presented. In particular, smoothing, histogram calculations, and the two-dimensional FFT are examined. Using these examples, the potential improvement a system such as PASM can provide over serial machines is demonstrated.

II. PARALLEL COMPUTATION UNIT

A. PCU Organization

The Parallel Computation Unit (PCU) contains processors, memories and an interconnection network. One configuration of these components is to connect a memory module to each processor to form a processor - memory pair called a processing element (PE). The interconnection network is used for communications between PEs. This configuration is called PE-to-PE and is shown in Figure 2. A pair of memory units is used for each memory module. This double-buffering scheme allows data to be moved between one memory unit and secondary storage (the Memory Storage System) while the processor operates on data in the other memory unit. In the PE-to-PE configuration, memory references are relatively fast, however the transfer of large blocks of data from processor to processor is delayed by the memory fetching and storing which must be done.

The P-to-M (processor-to-memory) configuration, shown in Figure 3, uses the interconnection network to connect the processors to the memory modules. Again, double-buffering is employed. With the P-to-M structure, every memory reference must travel through the interconnection network. To fetch an operand from memory, the processor must first send the address of the operand through the interconnection network to a memory. Then, the processor receives the operand from the memory via the interconnection network. Advantages of the P-to-M configuration are that a memory connected to a processor can be reconnected to another processor, effectively transfering the entire contents of the memory from one processor to another, and that the number of memories does not have to be equal to the number of processors (e.g. BSP [9]). A disadvantage is that all memory references must go through the interconnection network.

A more detailed analysis reveals some of the tradeoffs involved in these two configuations. If T_{mr} is the time required for a memory access (either a read or a write), and T_{in} is the time to send a data item through the interconnection network, then the time required for a memory reference in the P-to-M configuration, T_{P-M}, is given by

(1) $$T_{P-M} = T_{in} + T_{mr} + T_{in}.$$

T_{in} must be included twice, once for the processor to send the address to the memory and once for transferring the data. (The time required for controlling the network is omitted since control methods vary.)

For the PE-to-PE configuration the time required for a memory reference, T_{PE}, depends on the location of the memory which is to be used. If the memory is local, then

(3) $$T_{PE} = T_{mr}.$$

If the memory is connected to some other processor, then

(3) $$T_{PE} = T_{in} + T'_{mr} + T_{in}.$$

Again T_{in} is included twice. In this case, the first T_{in} represents the time required to transfer the address from the PE requesting the data item to the PE which has the data item. T'_{mr} represents the time required for the PE which has the data item to recognize and service the data request. This may require a significantly longer delay than T_{mr}. The second T_{in} is the time required to transfer the data item. If p is the probability of a local memory reference, then (2) and (3) can be combined to give the expected memory reference time

(4) $$E[T_{PE}] = pT_{mr} + (1-p)(T_{in} + T'_{mr} + T_{in}).$$

Comparing (1) and (4),

(5) $$T_{P-M} \geq E[T_{PE}]$$

for p sufficiently large. Thus, the "best" configuration is task dependent.

When operating in SIMD mode with the PE-to-PE configuration, it is often possible to omit one occurrence of T_{in} in (3) and reduce T'_{mr} to T_{mr}. This is done by computing the address of the desired data in the processor connected to the memory to be accessed. Thus, (4) reduces to

(6) $$E[T_{PE}] = pT_{mr} + (1-p)(T_{mr}+T_{in}).$$

Therefore, when operating in SIMD mode the PE-to-PE configuration is preferable.

When operating in MIMD mode, the PE-to-PE configuration requires that two processors be involved in every non-local memory reference. The efforts of two processors involved in a data transfer can be co-ordinated by having the processor which initiates the transfer interrupt the other processor or by dedicating one of these processors to handling data transfers. In the P-to-M configuration, the memories are shared by the processors, i.e., more than one processor can access the same memory for either data or instructions. However, for the image processing tasks that have been examined, most data and instructions can be stored in the local memory, reducing the impact of this consideration.

The PE-to-PE configuration will be used in PASM. Depending on the application for which a different partitionable SIMD/MIMD system is intended, the P-to-M configuration may be preferable. The interconnection networks, control structure, and secondary memory system described below can be used in conjunction with either.

B. Interconnection Networks

One of the major problems in designing parallel processing systems is the construction of an interconnection network to provide interprocessor communications. In this section a variety of recirculating (single stage) and multistage networks are described. Each of the networks discussed is compatible with the control and memory management schemes presented later. Thus, depending on the intended applications of a particular partitionable SIMD/MIMD machine being designed, any one of these networks can be used with into the overall system structure described in this paper.

Formally, an _interconnection network_ is a set of interconnection functions [36]. Each _interconnection function_ is a bijection (permutation) on the set of _N_ input/output addresses, the integers from 0 to N-1. The interconnection function f connects input i to output f(i), $0 \leq i < N$. Several types of interconnection networks are discussed below, using $p_{n-1} \cdots p_1 p_0$ to denote the binary representation of an arbitrary input/output address, $\bar{p}_i$ to denote the complement of p_i, and $\underline{n} = \log_2 N$.

The _Cube network_ consists of the n functions defined by:

$$\text{cube}_i(p_{n-1} \cdots p_{i+1} p_i p_{i-1} \cdots p_0) = p_{n-1} \cdots p_{i+1} \bar{p}_i p_{i-1} \cdots p_0$$

for $0 \leq i < n$ [36]. Typically, a multistage Cube network consists of n stages of switches, where stage i implements the cube_i interconnection function. The topology of the _indirect binary n-cube_ network [29] is shown in Figure 4. The switches, called interchange boxes, can be in either the straight or exchange state as shown. Each interchange box is independently controlled. Information on the capabilities of this network can be found in [29]. The STARAN flip network [4,5] is also based on the Cube interconnection functions and its capabilities are a subset of those of the binary n-cube [50].

The _Omega_ network [20], shown in Figure 5, is a multistage implementation of the "Shuffle-Exchange" [58]. In this network, an interchange box can be in either the straight, exchange, lower broadcast, or upper broadcast state as shown. Each interchange box is independently controlled. As in the indirect binary n-cube, stage i of the Omega network can implement cube_i

[40,50]. The order of the stages of these two networks is reversed, but for the following discussion on partitioning these differences are irrelevant, and both will be referred to as multistage Cube networks.

Consider partitioning a Cube network into 2^{n-r} subnetworks, each of which has 2^r inputs and outputs. This can be accomplished by grouping all of the input and output lines which agree in of their address bits. For example, consider grouping together all input and output lines which have the same n-r most significant address bits into a partition. To implement this, force the interchange boxes of the r through n-1 stages to be in the straight state. This prevents data transfers between processors whose addresses differ in the n-r most significant bits, thus preventing any communication between partitions [50]. Each subnetwork has the same properties as the whole network, so any subnetwork can be further subdivided into smaller partitions. Thus, varying sizes of partitions which are powers of two can be formed [53]. For example, for N=16, one partitioning of the network is into subnetworks of sizes 8, 4, 2, and 2.

A recirculating (single stage) network based on the Cube interconnection network can be modeled using the input and output selectors in Figure 6 [40]. Conceptually, for each x, $0 \leq x < n$, input selector x is connected to output selector $\text{cube}_i(x)$, $0 \leq i < n$, and output selector x is connected to input selector $\text{cube}_i(x)$, $0 \leq i < n$. Each pass through the network allows the implementation of one Cube function. In this implementation, partitioning is achieved by preventing the input selectors from using all of their output lines, and by preventing the output selectors from using all of their input lines. That is, if, for example, all of the addresses of the processors in a partition agree in the n-r most significant bits, then these processors should be prevented from using the input and output lines which implement the cube_{n-1} through cube_r interconnection functions.

The Plus-Minus 2^i (PM2I) network consists of the $2n-1$ interconnection functions defined by:

$$PM2_{+i}(j) = j+2^i \text{ modulo } N$$
$$PM2_{-i}(j) = j-2^i \text{ modulo } N$$

for $0 \leq i < n$, $0 \leq j < N$ [36]. Note that $PM2_{+(n-1)} = PM2_{-(n-1)}$. Feng's data manipulator network [11] consists of n stages of connections, where stage i implements the two functions $PM2_{+i}$ and $PM2_{-i}$, as shown in Figure 7. Each stage is controlled by only a pair of signals. An augmented data manipulator (ADM) network is a data manipulator with individual switch control, i.e., each switch can get any of the signals H (straight), U ($PM2_{-i}$), and D ($PM2_{+i}$). The capabilities of the ADM are a superset of those of the Omega network [40,50].

Like the Cube network, the ADM network can be partitioned into subnetworks of varying sizes which are powers of two [53]. In this case, to form 2^{n-r} independent subnetworks of size 2^r, the input and output lines which have the same n-r least significant address bits are grouped into a partition. This is implemented by forcing the switches of stages 0 through n-r into the H state. Since each subnetwork has the properties of the whole network, any subnetwork can be further subdivided into smaller partitions.

A recirculating network based on the PM2I functions can be modeled using input and output selectors shown in Figure 6. In this case, for each x, $0 \leq x < n$, input selector x is connected to output selector $PM2_{+i}(x)$ and $PM2_{-i}(x)$, $0 \leq i < n$, and output selector x is connected to input selector $PM2_{-i}(x)$ and $PM2_{+i}(x)$, $0 \leq i < n$. This single stage network can be partitioned in a manner similar to the multistage ADM network [53].

The multistage networks described above can be constructed in two ways: as combinational logic or as a pipeline [53]. In the pipeline construction, registers are placed between stages, increasing the speed of block transfers. In the following sections, it will be assumed that the processors will be partitioned such that their addresses agree in the low-order bit positions, so that either PM2I or Cube type networks can be used. Thus, any of the partitionable networks discussed above can be used in the PCU. For PASM, the multistage pipelined ADM network is being considered because of its speed and flexibility. However, a multistage pipelined Cube network may be sufficient for the system's needs. The tradeoffs are currently under investigation. More information about these networks can be found in [28,36,39,41,42,50,51,53-55].

III. MICRO CONTROLLERS

A. Addressing Conventions

In order to have a partitionable system, some form of multiple control units must be provided. In PASM, this is done by having $\underline{Q}=\underline{2}^q$ Micro Controllers (MCs), physically addressed (numbered) from 0 to Q-1. Each MC controls N/Q PCU processors, as shown in Figure 8.

An MC is a microprogrammable microprocessor which is attached to a memory module. Each memory module consists of a pair of memory units, "A" and "B," so that memory loading and computations can be overlapped. In SIMD mode, each MC fetches instructions from its memory module, executing the control flow instructions (e.g. branches) and broadcasting the data processing instructions to its PCU processors. The physical addresses of the N/Q processors which are connected to an MC must all have the same low-order q bits so that the network can be partitioned. The value of these low-order q bits is the physical address of the MC. A virtual SIMD machine of size RN/Q, where $R = 2^r$ and $1 \leq R \leq Q$, is obtained by loading R MCs with the same instructions and synchronizing the MCs. The physical addresses of these MCs must have the same low-order q-r bits so that all of the PCU processors in the partition have the same low-order q-r physical address bits. Similarly, a virtual MIMD machine of size RN/Q is obtained by combining the efforts of the PCU processors associated with R MCs which have the same low-order q-r physical address bits. In MIMD mode, the MCs may be used to help coordinate the activities of their PCU processors.

In each partition the PCU processors and memory modules are assigned logical addresses. Given a virtual machine of size RN/Q, the processors and memory modules for this partition have logical addresses (numbers) 0 to (RN/Q) - 1, $R = 2^r$, $0 \leq r \leq q$. Assuming that the MCs have been assigned as described above, then the logical number of a processor or memory module is the high-order r+n-q bits of the physical number. Recall that all of the physical addresses of the processors in a partition must have the same q-r low-order bits. For example, for N = 1024, Q = 16, and R = 4, one allowable choice of processors to form a partition of size RN/Q is those whose physical addresses are 3, 7, 11, 15,...1023. The high-order r+n-q = 8 bits of these 10-bit physical addresses are 0, 1, 2, 3,...255, respectively. The value of the low-order q-r = 2 bits of all of these physical processor addresses is equal to three.

Similarly, the MCs assigned to the partition are logically numbered (addressed) from 0 to R-1. For R > 1, the logical number of an MC is the high-order r bits of its physical number. Recall all of the physical addresses of the MCs in a partition must agree in the low-order q-r bits. For R = 1, there is only one MC and it is considered logical number 0. For example, if N = 1024, Q = 16, and R = 4, one allowable choice of four MCs is those whose physical addresses are 3, 7, 11, and 15. The high-order r = 2 bits of these four bit physical addresses are 0, 1, 2, and 3, respectively. The value of the low-order q-r = 2 bits of all the physical MC addresses is equal to three.

The PASM language compilers and operating system are used to convert from logical to physical addresses. Thus, a system user deals only with logical addresses.

B. Communications with the System Control Unit

When large SIMD jobs are run, that is jobs which require more than N/Q processors, more than one MC executes the same set of instructions. Since each MC has its own memory, if more than one MC is to be used, then several memories must be loaded with the same set of instructions. Another occasion which requires several MC memories to contain the same instructions is when the same program is to be run for several different sets of data. For example, suppose Q different images are to be processed using a program that requires N/Q processors for each image. Each MC will be executing the same program, but each will be working on a different image.

The fastest way to load several MC memories with the same set of instructions is to load all of the memories at the same time. This can be accomplished by connecting the Control Storage to all of the MC memory modules via a bus as shown in Figure 9. Each memory module is either enabled or disabled for loading from Control Storage, depending on the contents of a Q bit mask register called the Micro Controller Memory Load (MCML) register. MC memory module i is enabled if the i^{th} bit of the MCML register is a "1," otherwise it is disabled. The Q bit Micro Controller Memory Select (MCMS) register selects which memory unit an MC processor should use for instructions. A "1" in the ith bit means MC processor i is to use its A memory unit, while a "0" in the ith bit means it is to use its B memory unit. An enabled memory unit not being used by an MC processor receives the data from Control Storage.

The Q bit Micro Controller Status (MCS) register contains the go/done status of the MCs. A "0" in the ith bit means that MC i is done. When the ith bit is set to a "1," the MC sets its program counter to zero and begins executing or broadcasting the contents of the memory unit that is specified by the MCMS register. When the MC is done, it sets its bit of the MCS register to "0" and sends an interrupt to the System Control Unit to inform it that an MC has finished.

C. Communications Among Micro Controllers

Instructions which examine the collective status of all of the PEs of a virtual SIMD machine include "if any," "if all," and "if none." These instructions change the flow of control of the program at execution time depending on whether not any or all processors in the SIMD machine satisfy some condition. For example, if each PE is processing data from a different radar unit, but all PEs are looking for enemy planes, it is desirable for the control unit to know "if any" of the PEs has discovered a possible attack.

The task of computing an "if any" type of statement that involves several MCs can be handled using the existing hardware. This can be accomplished by having the PCU processors associated with these MCs use a recursive doubling type of algorithm.

Specialized hardware to handle this task can be added inexpensively and will result in a faster solution than the software approach. This approach uses a bus which connects the MCs and requires that each MC have access to the job identification number (ID) for the job it is running. The ID for a job on an MC may range from 0 to 2Q-1 since there can be at most 2Q jobs, i.e., one in each of the 2Q memory units. The Micro Controller Communication Bus (MCCB) consists of an ID bus and a data bus. When an "if any" type instruction is encountered, each MC associated with the job sends a request to the bus controller to use the communication bus. When one of the MCs becomes the first item in the queue the bus controller sends that MC a "permission to use the bus" signal so the MC may broadcast its job ID to all of the MCs (including itself) via the q+1 bit MCCB ID bus. If an MC is running the job with the ID which is on the ID bus it then puts its local results onto the MCCB data bus. The MCCB data bus is one bit wide and will be constructed using "wired and" technology, i.e., the bus is a Q input "wired and" gate. This allows all of the MCs associated with a job to put their data on the bus simultaneously. Then, while all this information is on the bus, all of the MCs associated with the job read the data and take the appropriate action. Each MC serviced removes itself from the MCCB queue.

For example, in the case of the "if any" instruction, when the job ID appears on the ID bus each MC puts its local results on the data bus. A "1" is sent if none of its PCU processors met the condition, a "0" is sent if any of its PCU processors met the condition. An MC which does not match the job ID will present a "1" to the data bus. If any PCU processor running the job meets the condition the data bus will be "0," however if no PCU processor meets the condition the bus will be "1." All of the MCs will then have access to this information, which will be needed to execute the conditional branch in the common instruction stream.

The hardware required to interface each MC to the MCCB is shown in Figure 10. The job ID is transmitted from a single MC to the MCCB ID bus by the tristate buffer only when that MC is given permission to use the MCCB. Then, each MC uses its comparator to compare its job ID to the MCCB ID bus.

A major problem associated with any bus in a multiprocessing system is contention. The MCCB is allocated on a priority basis. A priority ring based on the physical addresses of the MCs is used. MC i+1 modulo Q has a higher priority than MC i, $0 \leq i < Q$. The priority ring is broken by a Q bit shift register which contains exactly one "1." A "1" in the ith bit of the shift register indicates that MC i-1 modulo Q has the highest priority, and therefore MC i has the lowest priority. After each bus cycle the shift register is circularly shifted one bit, such that if MC i had the lowest priority it is given the highest priority for the next bus cycle. Details and a simulation are in [47].

The queue for the communication bus simply consists of a Q bit register which is loaded at the end of each bus cycle. A "1" in the ith bit of the register indicates that MC i is presently waiting to use the communication bus. After an MC has finished using the bus it resets its corresponding bit to zero, thus removing itself from the queue. A block diagram of the MCCB controller is shown in Figure 11.

Thus, this relatively inexpensive hardware design will allow rapid execution of the "if any," "if all," "if none," etc., instructions without burdening the System Control Unit. In addition to their use in SIMD programs, these instructions can be used to aid in synchronizing and coordinating PCU processors in a virtual MIMD machine.

D. Communications with the PCU Processors

The processors used in the PCU are to be constructed from user microprogrammable "bit-slice" components, available from manufacturers such as Texas Instruments [65], Advanced Micro Devices [2], and Intel [17]. Bit-slice components are designed such that several sets of these components can be combined to form processors of arbitrary word length. Bit-slice processor components typically include a computational unit, a sequencer, and special hardware to allow such features as pipelining and lookahead addition. The computational unit contains the processor's registers, the mechanism for register transfers, and the mechanism for arithmetic computation. The sequencer controls microprogram execution by calculating the next execution address. The choice of bit-slice processors over single chip microprocessors at this point in time is due for the most part to the advantages of speed and versatility seen in being able to have a 16-bit user microprogrammable processor. By having user microprogrammable processors, the instruction set can be optimized for parallel image processing.

The unique structure of PASM dictates that the processors used in the PCU be unique themselves. Since bit-slice processors are microprogrammable by the user they can be made to function in some special ways. The term _micro-function_ refers to the set of tasks carried out by applying one control word to the computational unit. Typical bit-slice processors, such as Texas Instrument's SN74S481 or Intel's 3000, offer a set of commonly used micro-functions as the only micro-functions available to the user. Micro-function instruction words can be represented by fewer bits than the actual control word if the set of available micro-functions is limited. For example, the Texas Instrument's SN74S481 has an 11-bit function word and a 24-bit control word. The micro-functions are translated into control words by a programmed logic array. The microprogram of the processor can then be stored more efficiently, the number of interchip connections is kept at a minimum, and the job of writing microcode can be compared with that of writing macrocode. The size of the microprogram store and the number of interchip connections become especially important when considering the construction of a large array of processors. The cost of limiting the set of available micro-functions is that of limiting overall versatility. However, most bit-slice processors offer a set of micro-functions complete enough for almost all practical purposes. If the micro-functions provided are not sufficient for PASM, then a processor which allows a customized encoding of control words into an allowable set of micro-function instruction words may be used.

When PASM is operating in SIMD mode, many processors are executing the same instruction stream in a synchronous fashion. If the MCs handle instruction decoding and sequencing for the sets of PCU processors under their control, as opposed to letting each PCU processor handle its own instruction decoding and sequencing, the number of duplicate control memory stores and sequencing hardware chips is reduced by a factor of N/Q.

For a set of PCU processors to operate in MIMD mode, the above scheme is not sufficient. The reason for this stems from the fact that each PCU processor in this set may execute a different instruction at the same time. Thus, the above scheme must be modified so that a subset of the PCU processors is capable of handling its own instruction decoding and sequencing while in MIMD mode and of allowing MCs to do these tasks while in SIMD mode. The size of such a set of processors would be dictated by the intended application of the system in question. Initial studies for PASM indicate that SIMD mode will be used much more often than MIMD mode and that MIMD tasks will, in general, require fewer processors. It may, therefore, be reasonable to expect that the size of the set of "dual mode" processors could be much smaller than the overall size of the PCU without significantly hindering computational capability. Furthermore, since the processors are microprogrammable it is possible for "dual mode" processors to have two instruction sets, one for MIMD mode and one for SIMD mode. The instruction sets are discussed further in [46].

E. Enabling and Disabling PCU Processors

In SIMD mode, all of the active PCU processors will execute instructions broadcast to them by their MC. A *masking scheme* is a method for determining which PCU processors will be active at a given point in time. An SIMD machine may have several different masking schemes. The masking scheme provides the system user with a device to enable some processors and disable others.

The *general masking* scheme uses an N-bit vector to determine which PCU processors to activate. Processor number i will be active if the i^{th} bit of the vector is a 1, for $0 \leq i < N$ (where the low order bit is the 0^{th}). For example, if N = 8 and the bit vector is 00101011, then only processors 0, 1, 3, and 5 will be active. Obviously, a general mask can activate any set of processors. These masks are specified in the SIMD program, and are part of the instruction stream broadcast by the control unit. A mask instruction is executed whenever a change in the active status of the processors is required. The Illiac IV, which has 64 processors and 64-bit words, uses general masks [57]. However, when N is larger, say 1024, a scheme such as this, where the mask size is N bits, becomes less appealing in terms of the difficulty in constructing and storing each 1024-bit mask.

The PE address masking scheme, introduced in [36], uses an n-position mask to specify which of the N PCU processors are to be activated. Each position of the mask corresponds to a bit position in the logical addresses of the processors. Each position of the mask will contain either a 0, 1, or X ("don't care") and the only processors that will be active are those whose address matches the mask: 0 matches 0, 1 matches 1, and either 0 or 1 matches X. Superscripts are repetition factors; square brackets denote a mask. For example, $[X^{n-1}0]$ activates all even numbered processors.

A negative PE address mask is similar to a regular PE address mask, except that it activates all those processors which do not match the mask. Negative PE address masks are prefixed with a minus sign to distinguish them from regular PE address masks. This type of mask, introduced in [37], can activate sets of processors a single regular PE address mask cannot, e.g., $[-0^n]$ activates all processors except for number 0.

The way in which PE address masks interact with various interconnection networks is analyzed in [36]. In [42] PE address masks are used to write SIMD machine algorithms. Other properties of these masks are discussed in [37,47,48].

Like general masks, PE address masks are specified in the SIMD program. PE address masks are more restrictive than general masks, in that a general mask can activate any arbitrary set of processors and a PE address mask cannot. However, for $N >> 64$, general masks are impractical in terms of storage requirements and ease of programming, and so system architects must consider alternatives. Together, regular and negative PE address masks provide enough flexibility to allow the easy programming of a variety of image processing tasks.

For ease of encoding and decoding, two bits are used to represent each PE address mask position. Figure 12 shows the mask word format for each MC, when $N = 1024$. The mask word consists of $2n+1 = 21$ bits, which allows masks having up to $n = 10$ positions and a sign bit to be specified. S is the sign bit for the mask. Of the remaining 20 bits, the $2(n-q) = 12$ high-order bits pertain to the PCU processors in an MC group, while the low-order $2q = 8$ bits pertain to the MC addresses. The entire mask word is a <u>physical mask</u> and is used with the physical addresses of the PCU processors and the MCs. A <u>logical mask</u> is a subset of the physical mask and consists of only that part of the mask needed to control the processors in a partition. The <u>Mask Decoder</u> of Figure 13 transforms the 21-bit mask word into a 64-bit general mask vector, for $N = 1024$ and $Q = 16$. A Mask Decoder is part of each MC. Each logical PE address mask is treated as the high-order portion of the physical mask. The portion of the physical mask not considered part of the logical mask is initialized with "X"s and left unaltered by program execution. The sign bit of the logical and physical masks is the same. The 64-bit general mask vector generated is for the 64 PCU processors associated with that MC. This is all that is required to translate a logical mask into enable signals.

The system of Figure 14 allows 64-bits of an arbitrary general mask vector or the output of the PE address Mask Decoder to be sent to the <u>Mask Vector Register</u> of each MC. If the general masking scheme were implemented on PASM, a reformatting operation would be needed. To execute a mask instruction, the MCs do not broadcast the entire mask to each processor, but transfer to a processor only the one bit of the mask that pertains to that processor. Assume the mask is being used in a partition of size $2^p=RN/Q$, where $n-q \leq p \leq n$. Due to the fact that the physical addresses of all of the PCU processors in a partition must agree in their n-p low-order bit positions, the system compiler must rearrange the programmer's logical general mask to form a physical general mask. This physical mask will be in a form which can be used more readily by the MCs than can the logical mask. The PCU processor whose logical address is $Rj + i$ will be the j^{th} processor controlled by the i^{th} logical MC, $0 \leq i < R$, $0 \leq j < N/Q$. The logical mask bit $Rj + i$ is moved to the physical mask position $(N/Q)i + j$. Then the MC whose logical address is i will load its N/Q-bit mask register with bits $i*N/Q$ through $(i+1)*N/Q-1$ of the physical mask. (The translation of logical MC addresses to physical addresses was described in section III.A.) This method of loading will send the i^{th} bit of the logical general mask to the i^{th} logical PCU processor.

As a compromise between the flexibility of general mask vectors and the conciseness of PE address masks, the MC is allowed to fetch the 64-bit vector from the Mask Vector Register, perform various logical operations on the vector, and then reload it. A logical OR of two (or more) vectors generated by PE address masks is equivalent to taking the union of the sets of processors activated by the masks. A logical AND is equivalent to the intersection. The complement operation can be used to implement negative PE address masks instead of using the exclusive-or gates shown in Figure 13.

Thus, PE address masks are concise, easily converted from logical to physical masks, and can be combined using boolean functions for additional flexibility. Examples of how PE address masks can be used in algorithms for image processing tasks are presented in section V.

Both general mask vectors and PE address masks are set at compile time. Data conditional masks will be implemented in PASM for use when the decision to enable and disable PEs is made at execution time.

Data conditional masks are the implicit result of performing a conditional branch dependent on local data in an SIMD machine environment, where the result of different PEs' evaluations may differ. As a result of a conditional *where statement* of the form

where <data condition> *do* ... *elsewhere* ...

each PE will set its own internal flag to activate itself for either the "do" or the "elsewhere," but not both. The execution of the "elsewhere" statements must follow the "do" statements; i.e., the "do" and "elsewhere" statements cannot be executed simultaneously. For example, as a result of executing the statement:

where $A > B$ *do* $C \leftarrow A$ *elsewhere* $C \leftarrow B$

each PE will load its C register with the maximum of its A and B registers, i.e., some PEs will execute "$C \leftarrow A$," and then the rest will execute "$C \leftarrow B$." This type of masking is used in such machines as the Illiac IV [3] and PEPE [10]. "Where statements" can be nested using a run-time control stack. This is discussed in [46].

IV. SECONDARY MEMORY SYSTEM

A. Introduction

The Memory Management System in PASM will have its own intelligence and will use the parallel secondary storage devices of the Memory Storage System. Giving the Memory Management System its own intelligence will help prevent the System Control Unit from being overburdened. The parallel secondary storage devices will allow fast loading and unloading of the N double-buffered PCU memory modules and will provide storage for system image and picture data and MIMD programs.

B. Memory Storage System

Secondary storage for the PCU memory modules is provided by the Memory Storage System. The Memory Storage System will consist of N/Q independent Memory Storage units, where N is the number of PCU memory modules and Q is the number of MCs. The Memory Storage units will be numbered from 0 to (N/Q)-1. Each Memory Storage unit is connected to Q PCU memory units. For $0 \leq i < N/Q$, Memory Storage unit i is connected to those memory modules whose physical addresses are of the form $(Q * i) + k$, $0 \leq k < Q$. Recall that, for $0 \leq k < Q$, MC k is connected to those processors whose physical addresses are of the form $(Q * i) + k$, $0 \leq i < N/Q$. Thus, Memory Storage unit i is connected to the i^{th} processor/memory module pair of each MC. This is shown for N = 32 and Q = 4 in Figure 15.

The two main advantages of this approach for a partition of size N/Q are that (1) all of the memory modules can be loaded in parallel and (2) the data is directly available no matter which partition (MC group) is chosen. This is done by storing in Memory Storage unit i the data for a task which is to be loaded into the i^{th} memory module of the virtual machine of size N/Q, $0 \leq i < N/Q$. Memory Storage unit i is connected to the i^{th} memory module in each MC group (i.e., memory modules $Q * i$, $(Q * i) + 1$, $(Q * i) + 2, \ldots$). Thus, no matter which MC group of N/Q processors is chosen, the data from the i^{th} Memory Storage unit can be loaded into the i^{th} memory module of the virtual machine, for all i, $0 \leq i < N/Q$, simultaneously.

For example, in Figure 15, if the partition of size N/Q = 8 chosen consists of the processors connected to MC 2, then Memory Storage unit 0 will load memory module 2, 1 will load 6, 2 will load 10, etc. If instead MC 3's processors are chosen, Memory Storage unit 0 will load memory module 3, 1 will load 7, 2 will load 11, etc.

Thus, for virtual machines of size N/Q, this secondary storage scheme allows all N/Q memory modules to be loaded in one parallel block transfer. This same approach can be taken if only $(N/Q)/2^d$ distinct Memory Storage units are available, where $0 \leq d \leq n-q$. In this case, however, 2^d parallel block loads will be required instead of just one.

Consider the situation where a virtual machine of size RN/Q is desired, $1 \leq R \leq Q$, and there are N/Q Memory Storage units. In general, a task needing RN/Q processors, logically numbered 0 to RN/Q-1, will require R parallel block loads if the data for the memory module whose high-order n-q logical address bits equal i is loaded into Memory Storage unit i. This is true no matter which group of R MCs (which agree in their low-order q-r address bits) is chosen.

For example, consider Figure 15, where N = 32 and Q = 4. Assume that a virtual machine of size 16 is desired. The data for the memory modules whose logical addresses are 0 and 1 is loaded into Memory Storage unit 0, for memory modules 2 and 3 into unit 1, for memory modules 4 and 5 into unit 2, etc. Assume the partition of size 16 is chosen to consist of the processors connected to MCs 0 and 2 (i.e., all even physically numbered processors). Then the Memory Storage units first load memory modules physically addressed 0, 4, 8, 12, 16, 20, 24, and 28 (simultaneously), and then load memory modules 2, 6, 10, 14, 18, 22, 26, and 30 (simultaneously). Given this assignment of MCs, the PCU memory module whose physical address is 2 * i has logical address i, $0 \leq i < 16$. Assume the processors and memory modules associated with MCs 1 and 3 are chosen. First, memory modules physically addressed 1, 5, 9, 13, 17, 21, 25, and 29 are loaded simultaneously, and then modules 3, 7, 11, 15, 19, 23, 27, and 31 are loaded simultaneously. In this case, the memory module whose physical address is (2 * i) + 1 has logical address i, $0 \leq i < 16$. No matter which pair of MCs is chosen, only two parallel block loads are needed.

Thus, for a virtual machine of size RN/Q, this secondary storage scheme allows all RN/Q memory modules to be loaded in R parallel block transfers, $1 \leq R \leq Q$. If only $(N/Q)/2^d$ distinct Memory Storage units are available, $0 \leq d \leq n-q$, then $R * 2^d$ parallel block loads will be required instead of just R.

The actual devices that will be used as Memory Storage units will depend upon the speed requirements of the rest of PASM, cost constraints, and the state of the art of storage technology at implementation time. Possibilities to be investigated include disks, bubble memories, and CCDs.

C. Handling Large Data Sets

The Memory Management System makes use of the double-buffered arrangement of the memory modules to enhance system throughput. The scheduler, using information from the System Control Unit such as the number of PCU processors needed and maximum allowable run time, will sequence tasks waiting to execute [49]. Typically, all of the data for a task will be loaded into the appropriate memory units before execution begins. Then, while a processor is using one of its memory units, the Memory Management System can be unloading the other unit and then loading it for the next task. When the task currently executing completes, the PCU processor can switch to its other memory unit for doing the next task. Based on image processing and pattern recognition tasks which have been examined thus far, it appears that the use of double-buffering, the potentially large memory modules, the number of processors in each virtual machine, and the special purpose design of PASM make time sharing of the PCU processors and the use of conventional paging inappropriate.

There may be some cases where all of the data will not fit into the PCU memory space allocated. Assume a memory frame is the amount of space used in the PCU memory units for the storage of data from secondary storage for a particular task. There are tasks where many memory frames are to be processed by the same program (e.g., maximum likelihood classification of satellite data [62]). The double-buffered memory modules can be used so that as soon as the data in one memory unit is processed, the processor can switch to the other unit and continue executing the same program. When the processor is ready to switch memory units, it signals the Memory Management System that it has finished using the data in the memory unit to which it is currently connected. Hardware to provide this signaling capability can be provided in different ways, such as using interrupt lines from the processors or by using logic to check the address lines between the processor and its memory modules for a special address code. The processor checks a data identification tag to ensure that the new memory frame is available, and then switches memory units, assuming the data is present. The Memory Management System can then unload the "processed" memory unit and then load it with the next memory frame or next task. Such a scheme, however, requires some mechanism which can make variable length portions of programs or data sets (i.e., local data) stored in one unit of a memory module available to the other unit when the associated processor switches to access the next memory frame. This scheme would be used only when multiple memory frames are to be processed.

One method to do this maintains a copy of local data in both memory units associated with a given processor so that switching memory units does not alter the local variable storage associated with the processor. A possible hardware arrangement to implement this makes use of two characteristics of the PASM memory access requirements: (1) secondary memory will not be able to load a given memory unit at the maximum rate it can accept data, and (2) PCU processors will not often be able (or desire) to write to memory on successive memory cycles. Because of these two characteristics, processor stores to local variable storage locations in an active memory unit can be trapped by a bus interface register and stored in the inactive memory unit by stealing a cycle on the secondary memory bus. In essence, this technique makes use of the conventional store-through concept as described in [16,23]. An exception to the second characteristic mentioned above is multiple precision data. If 16 bit words are assumed, then for higher precision it may be desirable to use two or four words as a group. However, a simple buffering scheme can handle this possibility.

The method described is applicable to any system which allows its processing tasks to utilize several separate memories and which requires that identical copies of variable amounts of certain data be maintained in all memories so used. Further information about this scheme and a discussion of other possible methods for providing local data storage are presented in [43].

D. Altering Loading Sequences

To further increase the flexibility of PASM, a task may alter the sequence of data processed by it during execution. As an example, consider a task which is attempting to identify certain features in a series of images. The task might examine a visible spectrum copy of an image and, based on features identified in the image, choose to examine an infrared spectrum copy of the same image. Rather than burden the System Control Unit to perform data loading sequence alterations, the task is allowed to communicate directly with the Memory Management System.

In the case of an SIMD task, the associated MC(s) determines if changes are required in the data loading sequence for the task. If so, an MC specifies the nature of the changes and communicates them to the Memory Management System without involving the System Control Unit. Each MC has the ability to generate loading sequence changes through a PCU processor. For tasks which require R MCs ($1 \leq R \leq Q$) logically numbered 0 to R-1, logical MC 0 will handle loading sequence changes. MC 0 uses logical processor number 0 of the virtual machine to establish a control information list in logical PCU memory module 0. (The Q PCU processors which can possibly be logically numbered 0 in a virtual machine are physically numbered 0, 1, 2,...,Q-1.) This list specifies in a concise fashion the loading sequence alterations required. The MC initiates the transfer of this list to the Memory Management System by using logical processor 0 to write a pointer to the list into the highest addressable memory location of its memory module. Through the use of a simple address comparison, the write into this memory location generates an interrupt to the Memory Management System. The Memory Management System recognizes the interrupt as a request for a loading sequence change and determines which MC is making the request. The Memory Management System uses the list of control information (via the pointer provided) to determine the loading sequence changes required.

An alternative method of interrupt generation is to use an interrupt line from each of the Q possible logical PCU processor 0's to the Memory Management System. The method selected for interrupt generation will depend upon the interrupt capabilities of the microprocessor used in the PCU. While the loading sequence control information could be passed directly from the MCs to the Memory Management System, the length of the connections required may make implementation more difficult and costly.

One hardware scheme which can transfer the control information list from a PCU memory module to the Memory Management System is shown in Figure 16. The hardware system shown is based upon having the Memory Management System coordinate the recognition of processor interrupts and the associated transfers of control information lists from the memory modules. The interrupt recognition portion is handled by the Processor Interrupt Control Logic while the transfer of control information lists is handled by the Memory Module Access Control Logic. Suppose processor i establishes a control information list in one of its memory units and writes the pointer to the list into its highest addressable memory location, generating a signal to the Interrupt Control on Interrupt Request Line i, where $0 \leq i < Q$. The Interrupt Control signals the Memory Management System, which then uses the Access Control to read the control information list from memory module i. Finally, the Memory Management System signals the Interrupt Control to generate a signal on the Interrupt Accepted Line to processor i.

The same hardware arrangement described for SIMD tasks is used for MIMD tasks. With each group of N/Q MIMD processors, there is associated a memory supervisor which is logical processor 0 within the group. All processors associated with a given memory supervisor make requests for loading sequence changes through the memory supervisor, without involving the MCs or System Control Unit. This reduces System Control Unit contention problems and helps prevent the MC(s), possibly busy orchestrating the activities of the virtual MIMD machine, from becoming overburdened.

E. Memory Management System

A set of microprocessors is dedicated to performing the Memory Management System tasks in a distributed fashion, i.e., one processor handles Memory Storage System bus control, one handles the scheduling tasks, etc.

This distributed processing approach is chosen in order to provide the Memory Management System with a large amount of processing power at low cost. Requests coming from different devices can be handled simultaneously. In addition, dedicating specific microprocessors to certain tasks simplifies both the hardware and software required to perform each task.

The basic architecture of the Memory Management System is shown in Figure 17. A master processor coordinates the concurrent tasks executed by the slave processors. A shared memory approach is planned due to the need to share data such as task queues. To reduce contention for the shared memory, each processor uses a local ROM and RAM for storage of code and local data. In addition, if necessary the shared memory may be interleaved [23] to further reduce contention. The degree of interleaving desirable may be determined by simulation studies or queuing theory analysis [14] of the Memory Management System. The processors within the Memory Management System may be implemented using commercially available fixed instruction set microprocessors. The new generation of 16-bit processors [24,30,35,59] is particularly appropriate since many provide special hardware for operations such as locked increment and test, memory protection and management, and problem/supervisor state switching.

The division of tasks chosen is based on the main functions which the Memory Management System must perform, including: (1) generating tasks based on PCU memory module load/unload requests from the System Control Unit; (2) interrupt handling and generating tasks for data loading sequence changes requested by the PCU processors physically numbered 0 to Q-1 (see previous subsection); (3) scheduling of Memory Storage System data transfers; (4) control of input/output operations involving peripheral devices and the Memory Storage System; (5) maintenance of the Memory Management System file directory information; and (6) control of the Memory Storage System bus system.

Most Memory Management System operations will be initiated by the System Control Unit, so the master processor will communicate with the System Control Unit and perform the task spawning operations associated with its requests. PCU processor interrupt handling is assigned to one slave processor, which sends requests for loading sequence changes to the scheduler. Scheduling of Memory Management System data transfers is assigned to another dedicated slave processor since the scheduling of data transfers will be complex and time consuming if near optimal operation of this system is to be realized. One slave is devoted to input/output between the Memory Storage System and peripheral devices, such as magnetic tape units. It handles any communications with the peripheral devices and arranges access to the Memory Storage units. The control and maintenance of the Memory Management System file system may be assigned several slave processors since file location operations associated with N/Q secondary storage devices may exceed the processing capabilities of one processor. Alternatively, a microprocessor may be assigned to each Memory Storage unit for file directory maintenance (e.g. intelligent disks), with a single slave coordinating this activity. Finally, another slave processor is devoted to performing the operations associated with setting the Memory Storage System buses' control signals needed to connect each Memory Storage unit to the appropriate PCU memory module.

The hardware structure of the Memory Management System is such that additional slave processors can be added to perform tasks that are not considered to be part of the Memory Management System at this time. In an actual prototype Memory Management System, interfaces for additional slave processors would be provided to facilitate system expansion and the incorporation of new features into the Memory Management System.

V. IMAGE PROCESSING ON PASM

A. Introduction

In this section, some examples of how PASM can be used to expedite image processing tasks are presented. A high level language algorithm to smooth an image and build a histogram demonstrates some of the features of a programming language for PASM. Implementations of this algorithm and of a two-dimensional Fast Fourier Transform algorithm demonstrate the ways in which PASM's parallelism may be used to obtain significant reductions of execution time on computationally expensive tasks.

Ideally a high level language for image processing will allow algorithms for image processing tasks to be expressed easily. As an example, a high level language algorithm which first smooths an image and then builds a histogram is given in subsection B. The language constructs used are described in detail in [44]. The language being designed for PASM is a procedure based structured language which allows the use of index sets similar to those in TRANQUIL [1]. The characteristics of the image processing problem domain are being used to facilitate compiling. Analyses of image processing algorithms, such as those presented here, are being employed to identify efficient techniques for performing common tasks and to define storage allocation strategies. Work is currently being done on the design of an intelligent compiler which incorporates information about commonly used parallel processing techniques that appear in image processing tasks. In the next subsection, a high level language algorithm is presented, followed by a discussion of the implementation of the algorithm on PASM.

B. Smoothing and Histogram Algorithms

The algorithm "picture," shown in Figure 18, has "pixin" as an input image and "pixout" as an output image. Both pixin and pixout contain 512 by 512 pixels. Each point of pixin is an eight bit unsigned integer representing one of 256 possible gray levels. Each point in the smoothed image, pixout(i,j), has the average gray level of pixin(i,j) and its eight nearest neighbors, pixin(i,j+1), pixin(i,j-1), pixin(i-1,j), pixin(i+1,j), pixin(i-1,j-1), pixin(i-1,j+1), pixin(i+1,j+1), and pixin(i+1,j-1). Boundary points of pixout are not calculated since their corresponding points do not have eight adjacent neighbors. The "picture" routine also produces a 256 bin (one bin for each gray level) histogram, "hist," of the smoothed image.

Consider how this could be implemented on PASM, with N = 1024. Assume that the 1024 PEs are logically arranged as an array of 32 by 32 PEs, and that the PE addresses range from 0 to 1023:

```
PE 0    PE 1  . . .   PE 31
PE 32   PE 33 . . .   PE 63
                .
PE 992        . . .   PE 1023
```

Each PE stores a 16 by 16 block of the 512 by 512 image. Assume that the 16 by 16 blocks are stored in row major order. Thus, PE 0 stores the pixels in columns 0 to 15 of rows 0 to 15, PE 1 stores the pixels in columns 16 to 31 of rows 0 to 15, and so on. In general, $N = 2^n$ PEs operate on a picture of 2^k by 2^k pixels. The PEs are arranged as a $2^{n/2}$ by $2^{n/2}$ array, where n/2 is an integer, such that each PE stores in its memory a block of pixels of size $2^{k-(n/2)}$ by $2^{k-(n/2)}$.

For notational purposes, let each PE consider its 16 by 16 matrix as

$$H = \begin{vmatrix} h(0,0) & \ldots & h(0,14) & h(0,15) \\ & \ldots & & \\ h(15,0) & \ldots & & h(15,15) \end{vmatrix}.$$

Also, let the subscripts of h(i,j) extend to -1 and 16, in order to aid in calculations across boundaries of two adjacent blocks in different PEs. For example, the pixel to the left of h(0,0) is h(0,-1), and the pixel below h(15,15) is h(16,15). Therefore, $-1 \le i,j \le 16$.

A general algorithm to perform the smoothing on pixel h(i,j) to yield smoothed pixel hs(i,j) is:

```
for i ← 0 to 15 do
    for j ← 0 to 15 do
        hs(i,j) ← 1/9 * ( h(i+1,j) + h(i-1,j) + h(i,j+1) + h(i,j-1)
                  + h(i+1,j-1) + h(i+1,j+1) + h(i-1,j+1) + h(i-1,j-1) + h(i,j) )
```

The approach of this algorithm is to perform 1024 16 by 16 pixel evaluations in parallel rather then one 512 by 512 pixel evaluation as in the sequential algorithm.

At the boundaries of each 16 by 16 array, data must be transmitted between PEs in order to calculate the smoothed value, hs. For example, h(0,16) must be transferred from the PE "to the right of" the local PE, except for PEs 31,63,95,...,1023, those at the "right edge" of the logical array of PEs. (Pixels on the edges of the 512 by 512 array are not smoothed.) The necessary data transfers must be performed before the algorithm above is executed. The transfer of data from PE i+1 is illustrated as follows. h'(i,j) denotes an entry in the H matrix of PE i+1.

```
                 PE i                               PE i+1
h(0,0) . . .   h(0,14) h(0,15)          h(0,16) = h'(0,0)
h(1,0) . . .   h(1,14) h(1,15)          h(1,16) = h'(1,0)
         . . .                                . . .
h(15,0) . . . h(15,14) h(15,15)        h(15,16) = h'(15,0)
```

The compiler generated code for this transfer can be expressed as follows. "Set ICN to PE+j" sets the interconnection network so that PE P sends data to PE P + j modulo N, $0 \leq j < N$. PEs transfer data through Data Transfer Registers. The data are loaded into the DTRin of each PE, the TRANSFER command moves the data through the network, and the final data are retrieved from DTRout [53]. The command MASK [address set] is a PE address mask that determines which PEs will execute the instructions that follow. The absence of a mask implies all PEs are active. The mask $[-X^5 1^5]$ deactivates the PEs on the right side of the image, i.e., PEs 31,63,...,1023.

```
SET ICN to PE-1;
DTRin ← h(0 → 15,  0);
TRANSFER;
MASK [-X⁵1⁵] h(0 → 15, 16) ← DTRout;
```

The transfers of data for the remaining three sides of the array H, i.e., from PEs i-1, i+32, and i-32, are accomplished in a similar manner. Smoothing of the four points h(0,0), h(0,15), h(15,0), and h(15,15) require data that reside in PEs i-33, i-31, i+31, and i+33, respectively. This necessitates four additional parallel pixel transfers. Both the multistage Cube and PM2I networks can perform each of these connections in a single pass.

In order to perform a smoothing operation on a 512 by 512 image by the parallel smoothing of 256 point blocks of size 16 by 16, the total number of parallel word transfers is 4(16) + 4 = 68. The corresponding sequential algorithm needs no data transfers between PEs, but calculates hs for 512 * 512 = 262,144 points. If no data transfers were needed, the parallel algorithm would be faster than the sequential algorithm by a factor of 262,144/256 = 1024. If it is assumed that each parallel data transfer requires at most as much time as one smoothing operation, then the time factor is 262,144/324 = 809. That is, the parallel algorithm is about three orders of magnitude faster than the sequential algorithm. The approximation is a conservative one, since calculating the addresses of the nine pixels involves nine multiplications using the subscripts [15]. Block transfers of the data on the four sides of the array could be done rapidly with a pipelined multistage interconnection network [53]. (Another factor which must be considered in evaluating the speedup is processor speed. An IBM 370 will process data faster than a typical microprocessor. Even with possible differences in processing speed, PASM will still perform this task at least two orders of magnitude faster.)

Now consider implementing the histogram calculation. Since the image hs is spread out over 1024 PEs, each PE will calculate a 256 bin histogram based on its 16 by 16 segment of the image. Then these "local" histograms will be combined using the algorithm described below. This algorithm is demonstrated for N = 16 and B = 4 bins, instead of N = 1024 and B = 256 bins, in Figure 19. Both the multistage Cube and PM2I networks can perform each of the needed transfers in a single pass.

In the first $b = \log_2 B$ steps, each block of B PEs performs B simultaneous recursive doublings to compute the histogram for the portion of the image contained in the block. At the end of the b steps, each PE has one bin of this partial histogram. The way in which this is accomplished is by first dividing the B PEs of a block into two groups. Each group accumulates the sums for half of the bins, and sends the bins it is not accumulating to the group which is accumulating those bins. At each step of the algorithm, each group of PEs is divided in half such that the PEs with the lower addresses form one group, and the PEs with the higher addresses form another. For example, in Figure 19 in the first step, the group of PEs 4,5,6, and 7 is divided into the two groups of PEs 4 and 5, and 6 and 7. The accumulated sums are similarly divided in half based on their indices in the histogram. Next, the group with the lower PE addresses sends the group with the higher addresses the half of the sums which has the higher indices, while the group with the higher addresses sends the group with the lower addresses the sums with the lower indices. For example, in Figure 19, in the first step, PEs 4 and 5 send their bins 2 and 3 data to PEs 6 and 7. At the same time PEs 6 and 7 send their bins 0 and 1 data to PEs 4 and 5. After b steps, each PE has the total value for one bin from the portion of the image contained in the B PEs in its block.

The next $\log_2(N/B)$ steps combine the results for these blocks to yield the histogram of the entire image spread out over B PEs, with the sum for bin i in processor i, $0 \leq i < B$. This is done by performing $\log_2(N/B)$ steps of a recursive doubling algorithm to sum the partial histograms from the N/B blocks. This is shown by the last two steps of Figure 19. A general algorithm to compute the B bin histogram for an image spread out over N PEs is shown in Figure 20. For the "picture" algorithm example discussed above, N = 1024 and B = 256.

A sequential algorithm for calculating hist for a 512 by 512 picture requires 512 * 512 = 262,144 additions. The parallel algorithm described above uses 16 * 16 = 256 additions for each PE to calculate its "local" histogram, and 255 + 2 = 257 steps (transfer and add) to merge the histogram into the first 256 PEs. At step i in the computation of the partial histograms, $0 \leq i < \log_2 B$, the number of data transfers required is $B/(2^{i+1})$. A total of B-1 transfers are performed in the first $\log_2 B$ steps of the algorithm. $\mathrm{Log}_2(N/B)$ parallel transfers are needed to combine the partial histograms. In general, therefore, this technique requires $B-1+\log_2(N/B)$ parallel transfer/add operations, where $N \geq B$ and B is a power of two, plus the additions needed to compute the local histograms. For an M by M image spread out over N PEs, M a power of two, the number of parallel additions to compute the local histograms equals the number of pixels in each PE, which equals M^2/N. The number of addition steps is therefore reduced by a factor of N, at the added cost of $B-1+\log_2(N/B)$ transfer/add operations. The result of the algorithm, i.e., the histogram, is spread out over the first B processors. This distribution may be efficient for further processing on the histogram, e.g., finding the maximum or minimum, or for smoothing the histogram. If it is necessary for the entire histogram to be in a single processor, B-1 additional parallel data transfers are required.

C. Two-Dimensional FFT

As an example of the applicability of parallel computations to a different type of image processing task, the Fast Fourier Transform (FFT) is considered. A parallel algorithm for the one-dimensional FFT, often used in speech processing [52], is presented in [49]. Here the two-dimensional FFT is examined.

The two-dimensional discrete Fourier transform (DFT) of an L by M array of elements S(l,m) is defined as

$$F(v,w) = \sum_{l=0}^{L-1} \sum_{m=0}^{M-1} S(l,m)\, W_L^{vl}\, W_M^{wm} \qquad 0 \le v < L,\ 0 \le w < M$$

The two-dimensional transform can be decomposed in such a way as to reduce its computation to the execution of a number of one-dimensional DFTs. Performing the DFT on each row of the array yields

$$G(l,w) = \sum_{m=0}^{M-1} S(l,m)\, W_M^{wm} \qquad 0 \le l < L,\ 0 \le w < M$$

The DFT of the array can then be obtained by taking the DFT of each column of G:

$$F(v,w) = \sum_{l=0}^{L-1} G(l,w)\, W_L^{vl} \qquad 0 \le v < L,\ 0 \le w < M$$

Thus the two-dimensional transform can be obtained by computing L one-dimensional M-point transforms on the L rows of the S array, then computing M one-dimensional L-point transforms on the M columns of the G array resulting from the row transforms [27,31].

Consider performing the DFT on an M by M array S. For example, assume M = 1024, and the array represents a 1024 by 1024 point picture. Figure 21 outlines an efficient parallel two-dimensional FFT algorithm which uses M PEs. It is assumed that initially PE i contains row i of the array S. The DFTs on the rows of S are performed by executing a serial FFT in each PE, on the row of S contained in that PE. This serial FFT can be executed simultaneously by each of the M PEs. The resulting G array has row i in PE i. The transpose operation is performed on G, to rearrange the array so that PE i contains column i of G. A serial FFT executed in parallel in each PE performs the DFT on the columns of G, producing the transform array F.

Figure 22 presents an algorithm to transpose the array G. The basic operation performed is the transfer of array element G(j,k) from PE j to PE k. This is done for M G(j,k)'s in parallel, using an interconnection function which sends data from PE j to PE (j+i) mod M for all of the G(j,k) for which (k-j) mod M is equal to i. The parallel transfer operation is performed for $1 \leq i < M$. (For i=0, no transfer is needed; i.e., the diagonal of the transpose matrix is the same as the diagonal of the original matrix.) For each i value, the element which PE j sends is the kth element of the row of G held in PE j, where k=(j+i) mod M. That element, received in PE k, is stored as the jth element of the column of G transpose being created in PE k, where j=(k-i) mod M. In the algorithm, it is assumed that each PE has an address register, ADDRESS, which contains the integer i in PE i, $0 \leq i < M$. In each PE, G denotes the address of the first element of the G vector (row) that is stored in that PE; G^T denotes the address of the first element of the G transpose vector in each PE. If b bytes of storage are needed for each array element, G(j,k) is initially in location G+k*b of PE j , and after execution of the transpose algorithm, G(j,k) is in location G^T+j*b of PE k, $0 \leq j,k < M$. Each of the interprocessor data transfers needed for the transpose can be done in a single pass through either a multistage Cube or PM2I network.

The parallel two-dimensional FFT algorithm presented requires $M\log_2 M$ parallel complex multiplications and M-1 parallel data transfers to compute the two-dimensional DFT of an M by M array. The computation of the intermediate array G by broadcasting a serial FFT algorithm to all PEs, where each PE has a different row of S, entails $(M/2)\log_2 M$ parallel complex multiplications. Serial computation of the DFTs on the M rows of S would require $(M^2/2)\log_2 M$ complex multiplications, so speedup by a factor of M is achieved. This is the maximum speedup obtainable with M PEs. (Similarly, maximal reduction in the number of complex additions is achieved. In general, however, the multiplication time will dominate the addition time.) Computation of F from the transpose of G likewise entails $(M/2)\log_2 M$ parallel complex multiplications. M-1 parallel data transfers are required to transpose the G array. For M = 1024, a total of 10,240 parallel complex multiplications and 1023 parallel transfers will be executed. Serial implementation of the FFT computation would require $M^2\log_2 M$ complex multiplications, which for M = 1024 would be 10,485,760. In a system such as PASM, the FFT algorithm would be a system function, callable as a library routine from a user's program.

VI. CONCLUSIONS

PASM, a large scale partitionable SIMD/MIMD multimicroprocessor system for image processing and pattern recognition, has been presented. Its overall architecture was described and examples of how PASM can realize significant computational improvements over conventional systems was demonstrated. Current work includes specifying the hardware design details and developing the operating system and programming languages for a prototype system. A dynamically reconfigurable system such as PASM should be a valuable tool for both image processing/pattern recognition and parallel processing research.

Acknowledgements - The authors thank R. J. McMillen, P. A. England, J. Bogdanowicz, and M. Washburn for their contributions to the design of PASM.

VII. REFERENCES

1 Abel, N. E., et al., "TRANQUIL: a language for an array processing computer," AFIPS 1969 SJCC, May 1969, pp. 57-68.

2 Advanced Micro Devices, The AM2900 Family Data Book, 1976.

3 Barnes, G., et al., "The Illiac IV computer," IEEE Trans. Comp., Vol. C-17, No. 8, Aug. 1968, pp. 746-757.

4 Batcher, K. E., "The flip network in STARAN," 1976 Int'l Conf. Parallel Processing, Aug. 1976, pp. 65-71.

5 Batcher, K. E., "The multi-dimensional access memory in STARAN," IEEE Trans. Comp., Vol. C-26, Feb. 1977, pp. 174-177.

6 Bogdanowicz, J., and Siegel, H. J., "A partitionable multi-micropgrammable-microprocessor system for image processing," 1978 IEEE Computer Society Workshop on Pattern Recognition and Artificial Intelligence, Apr. 1978, pp. 141-144.

7 Bouknight, W. J., et al., "The Illiac IV system," Proc. IEEE, Vol. 60, Apr. 1972, pp. 369-388.

8 Briggs, F., Fu, K. S., Hwang, K. and Patel, J., "PM4 - a reconfigurable multimicroprocessor system for pattern recognition and image processing," Nat'l. Comp. Conf., June 1979, pp. 255-265.

9 Burroughs, BSP - Burroughs Scientific Processor, Burroughs Corp., June 1977.

10 Crane, B. A., et al., "PEPE computer architecture," Compcon 72, IEEE Comput. Soc. Conf., Sept. 1972, pp. 57-60.

11 Feng, T., "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comp., Vol. C-23, Mar. 1974, pp. 309-318.

12 Flynn, M. J., "Very high-speed computing systems," Proc. IEEE, Vol. 54, Dec. 1966, pp. 1901-1909.

13 Fu, K. S., "Special computer architectures for pattern recognition and image processing - an overview," Nat'l. Computer Conf., June 1978, pp. 1003-1013.

14 Fung, F. T. and Torng, H. C., "Analysis of memory conflicts in a multiple microprocessor system," IEEE Trans. Comp., Vol. C-28, No. 1, Jan. 1979, pp. 28-37.

15 Gries, D., Compiler Construction for Digital Computers, John Wiley & Sons, Inc., New York, N.Y., 1971.

16 Hayes, J., Computer Architecture and Organization, McGraw-Hill, New York, 1978.

17 Intel 3000 Data Sheets, Intel Corporation, Santa Clara, California 95051.

18 Keng, J., and Fu, K. S., "A special computer architecture for image processing," 1978 IEEE Comput. Soc. Conf. Pattern Recognition and Image Processing, June 1978, pp. 287-290.

19 Kruse, B., "A parallel picture processing machine," IEEE Trans. Comp., Vol. C-22, Dec. 1973, pp. 1075-1087.

20 Lawrie, D. H., "Access and alignment of data in array processor," IEEE Trans. Comp., Vol. C-24, No. 12, Dec. 1975, pp. 1145-1155.

21 Lipovski, G. J., "On a varistructured array of microprocessors," IEEE Trans. Comp., Vol. C-26, Feb. 1977, pp. 125-138.

22 Lipovski, G. J., and Tripathi, A., "A reconfigurable varistructure array processor," 1977 Int'l. Conf. Parallel Processing, Aug. 1977, pp. 165-174.

23 Matick, R. E., "Memory and storage," in Introduction to Computer Architecture, H. S. Stone, editor, Science Research Associates, Chicago,

Illinois, 1978.

24 Morse, S. P., Pohlman, W. B., and Ravenel, B. W., "The Intel 8086 microprocessor: a 16-bit evolution of the 8080," Computer, Vol. 11, No. 6, June 1978, pp. 18-27.

25 Nutt, G. J., "Microprocessor implementation of a parallel processor," 4th Symp. Comp. Arch., Mar. 1977, pp. 147-152.

26 Nutt, G. J., "A parallel processor operating system comparison," IEEE Trans. Software Engineering, Vol. SE-3, Nov. 1977, pp. 467-475.

27 Oppenheim, A. V. and Schafer, R. W., Digital Signal Processing, Prentice-Hall, Englewood Cliffs, N.J., 1975.

28 Patel, J. H., "Processor-memory interconnections for multiprocessors," 6th Int'l. Symp. Comp. Arch., Apr. 1979, pp. 168-177.

29 Pease, M. C., "The indirect binary n-cube microprocessor array," IEEE Trans. Comp., Vol. C-26, May 1977, pp. 458-473.

30 Peuto, B. L., "Architecture of a new microprocessor," Computer, Vol. 12, No. 2, Feb. 1979, pp. 10-21.

31 Rabiner, L. R. and Gold, B., Theory and Application of Digital Signal Processing, Prentice-Hall, Englewood Cliffs, N.J., 1975.

32 Reeves, A. P., and Rindfuss, R., "The BASE 8 binary array processor," 1979 IEEE Comp. Soc. Conf. Pattern Recog. and Image Processing, Aug. 1979, pp. 250-255.

33 Rohrbacker, D., and Potter, J. L., "Image processing with the Staran parallel computer," Computer, Vol. 10, Aug. 1977, pp. 54-59.

34 Ruben, S., Faiss, R., Lyon, J., and Quinn, M., "Application of a parallel processing computer in LACIE," 1976 Int'l. Conf. Parallel Processing, Aug. 1976, pp. 24-32.

35 Shima, M., "Two versions of 16-bit chip span microprocessor, microcomputer needs," Electronics, Vol. 51, No. 26, Dec. 1978, pp. 81-88.

36 Siegel, H. J., "Analysis techniques for SIMD machine interconnection networks and the effect of processor address masks," IEEE Trans. Comp., Vol. C-26, Feb. 1977, pp. 153-161.

37 Siegel, H. J., "Controlling the active/inactive status of SIMD machine processors," 1977 Int'l. Conf. Parallel Processing, Aug. 1977, p. 183.

38 Siegel, H. J., "Preliminary design of a versatile parallel image processing system," Third Biennial Conf. on Computing in Indiana, Apr. 1978, pp. 11-25.

39 Siegel, H. J., "Partitionable SIMD computer system interconnection network universality," 16th Annual Allerton Conf. on Communication, Control, and Computing, Oct. 1978, pp. 586-595.

40 Siegel, H. J., "Interconnection networks for SIMD machines," Computer, Vol. 12, June 1979, pp. 57-65.

41 Siegel, H. J., "Partitioning permutation networks: the underlying theory," 1979 Int'l. Conf. Parallel Processing, Aug. 1979, pp. 175-184.

42 Siegel, H. J., "A model of SIMD machines and a comparison of various interconnection networks," IEEE Trans. Comp., to appear Dec. 1979.

43 Siegel, H. J., Kemmerer, F., and Washburn, M., "Parallel memory system for a partitionable SIMD/MIMD machine," 1979 Int'l. Conf. Parallel Processing, Aug. 1979, pp. 212-221.

44 Siegel, H. J., McMillen, R. J., Mueller, P. T., Jr., and Smith, S. D., A Versatile Parallel Image Processor: Some Hardware and Software Problems, School of Electrical Engineering, Purdue University, Technical Report TR-EE 78-43, Oct. 1978.

45 Siegel, H. J., McMillen, R. J., and Mueller, P. T. Jr., "A survey of interconnection methods for reconfigurable parallel processing systems,"

Nat'l. Comp. Conf., June 1979, pp. 529-542.

46 Siegel, H. J., and Mueller, P. T., Jr., "The organization and language design of microprocessors for an SIMD/MIMD system," 2nd Rocky Mt. Symp. on Microcomputers, Aug. 1978, pp. 311-340.

47 Siegel, H. J., Mueller, P. T., Jr., and Smalley, H. E., Jr., Preliminary Design Alternatives for a Versatile Parallel Image Processor, School of Electrical Engineering, Purdue University, Technical Report TR-EE 78-32, June 1978.

48 Siegel, H. J., Mueller, P. T., Jr., and Smalley, H. E., Jr., "Control of a partitionable multimicroprocessor system," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 9-17.

49 Siegel, H. J., Siegel, L. J., McMillen, R. J., Mueller, P. T., Jr., and Smith, S. D., "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition," 1979 IEEE Comp. Soc. Conf. Pattern Recog. and Image Processing, Aug. 1979, pp. 214-224.

50 Siegel, H. J., and Smith, S. D., "Study of multistage SIMD interconnection networks," 5th Annual Symp. Comp. Arch., Apr. 1978, pp. 223-229.

51 Siegel, H. J., and Smith, S. D., "An interconnection network for multimicroprocessor emulator systems," 1st Int'l. Conf. Distributed Computing Systems, Oct. 1979.

52 Siegel, L. J., "Features for the identification of mixed excitation in speech analysis," 1979 IEEE Int. Conf. Acoustics, Speech, Signal Processing, Apr. 1979, pp. 752-755.

53 Smith, S. D., and Siegel, H. J., "Recirculating, pipelined, and multistage SIMD interconnection networks," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 206-214.

54 Smith, S. D. and Siegel, H. J., "An emulator network for SIMD machine interconnection networks," 6th Int'l. Symp. Comp. Arch., Apr. 1979, pp. 232-241.

55 Smith, S. D. and Siegel, H. J., The Design and Analysis of Interconnection Networks for Partitionable Parallel Processing Systems, School of Electrical Engineering, Purdue University, Technical Report TR-EE 79-39, Aug. 1979.

56 Stamopoulous, C. D., "Parallel algorithms for joining two points by a straight line segment," IEEE Trans. Comp., Vol. C-23, June 1974, pp. 642-646.

57 Stevens, K. G., Jr., "CFD - A FORTRAN-like language for the Illiac IV," Conf. Programming Languages and Compilers for Parallel and Vector Machines, ACM, Mar. 1975, pp. 72-76.

58 Stone, H. S., "Parallel processing and the perfect shuffle," IEEE Trans. Comp., Vol. C-20, Feb. 1971, pp. 153-161.

59 Stritter, E. and Gunter, T., "A microprocessor architecture for a changing world: the Motorola 68000," Computer, Vol. 12, No. 2, Feb. 1979, pp. 43-52.

60 Sullivan, H., Bashkow, T. R. and Klappholz, K., "A large scale homogeneous, fully distributed parallel machine," 4th Annual Symp. Comp. Arch., Mar. 1977, pp. 105-124.

61 Sullivan, H., et al., "The node kernel: resource management in a self-organizing parallel processor," 1977 Int'l. Conf. Parallel Processing, Aug. 1977, pp. 157-162.

62 Swain, P. H., Siegel, H. J., and Smith, B. W., "A method for classifying multispectral remote sensing data using context," Symp. on Machine Processing of Remote Sensing Data, June 1979, pp. 343-353.

63 Swan, R. J., Fuller, S. H., and Siewiorek, D. P., "Cm*: a modular,

multi-microprocessor," Nat'l. Computer Conf., June 1977, pp. 645-655.

64 Swan, R. J., et al., "The implementation of the Cm* multi-microprocessor," Nat'l. Computer Conf., June 1977, pp. 645-655.

65 Bipolar Microcomputer Components Data Book, Texas Instruments Inc., Dallas, Texas.

66 Wulf, W. A., Bell, C. G., "C.mmp - a multi-miniprocessor," AFIPS 1972 FJCC, Dec. 1972, pp. 765-777.

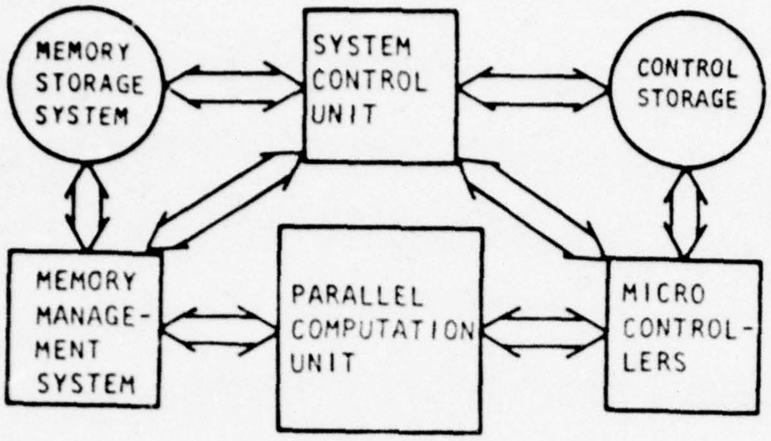


Fig. 1. Block diagram overview of PASM.

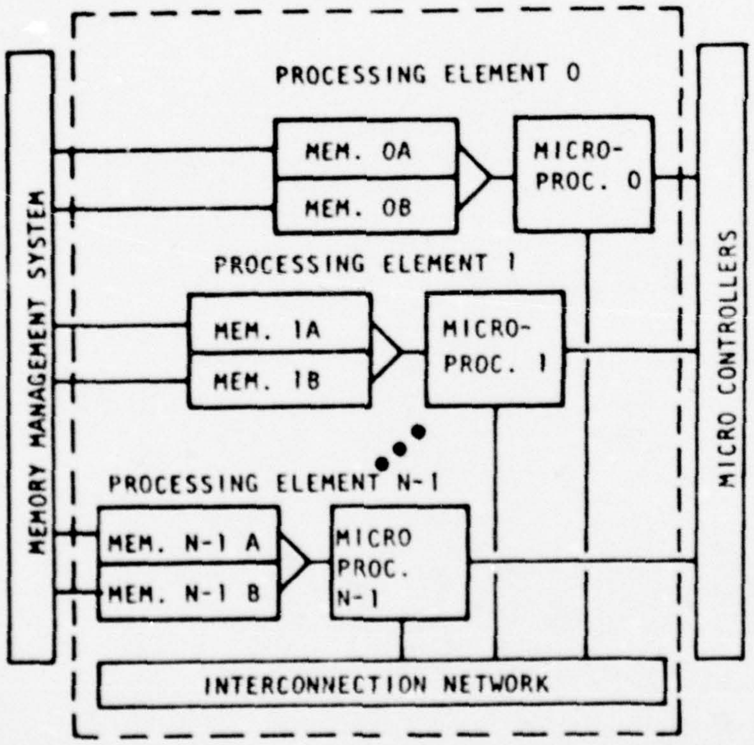


Fig. 2. PE-to-PE configuration of the Parallel Computation Unit.

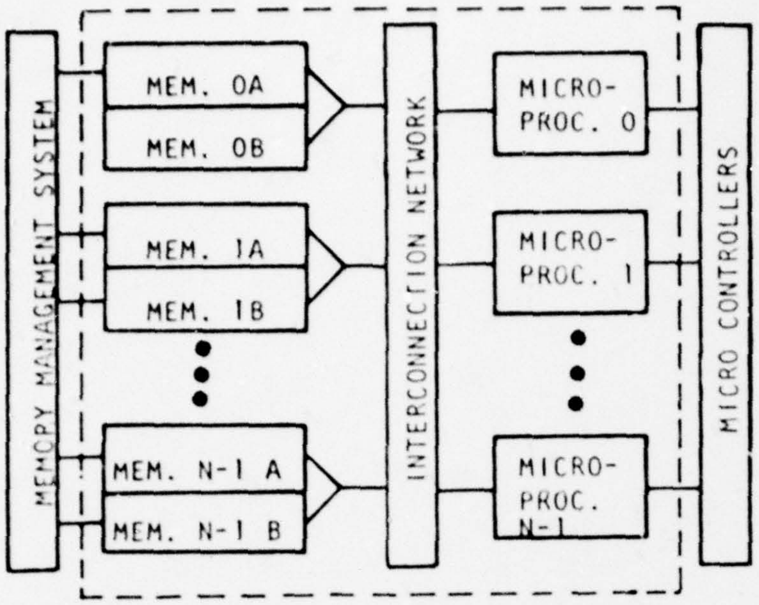


Fig. 3. Processor-to-memory configuration of the Parallel Computation Unit.

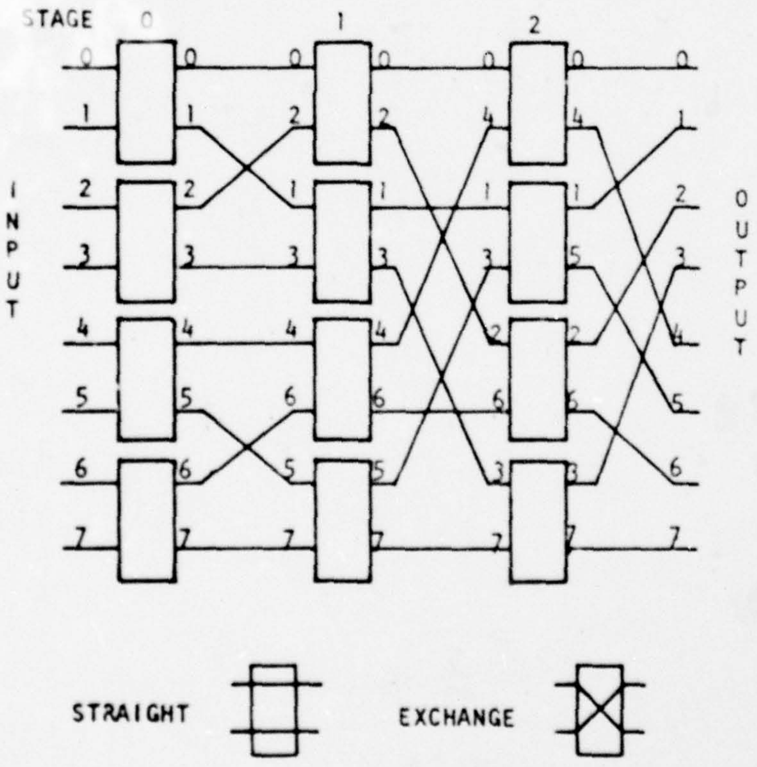


Fig. 4. Indirect binary n-cube network built with interchange boxes, for N=8.

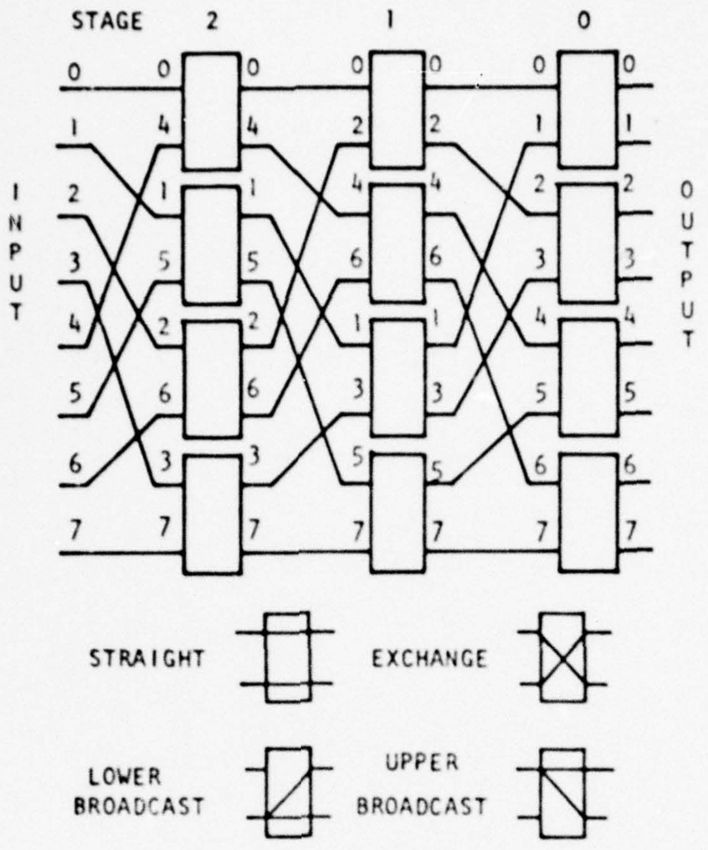


Fig. 5. Omega network built with interchange boxes, for N=8.

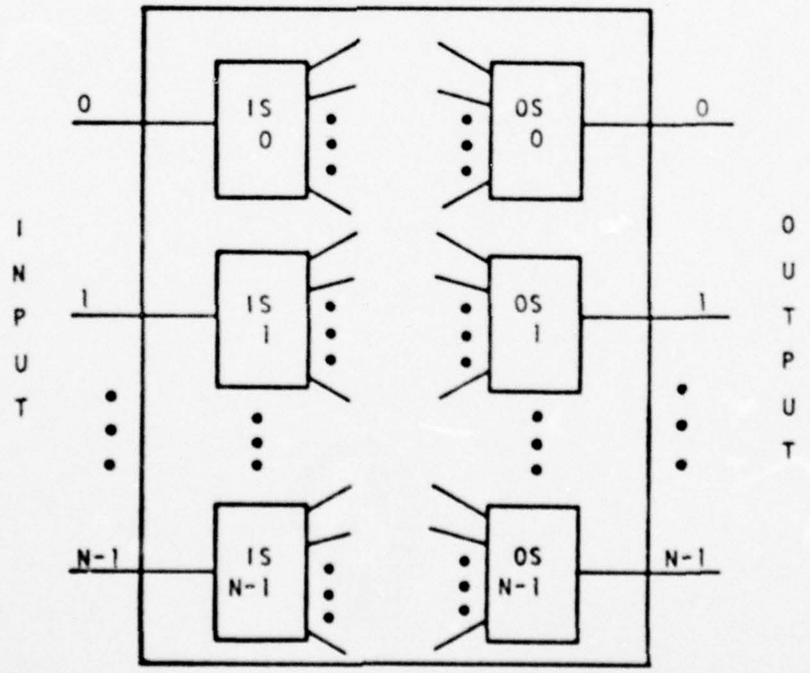


Fig. 6. Conceptual view of a recirculating (single stage) network. "IS" is input selector, "OS" is output selector.

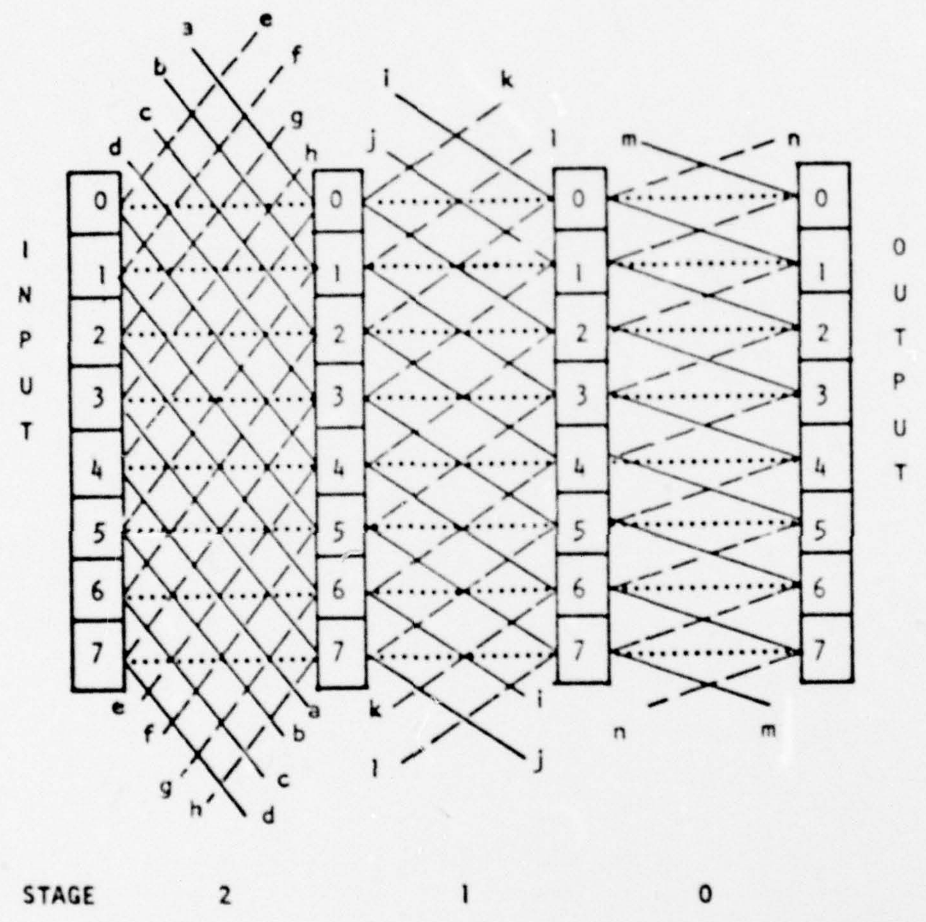


Fig. 7. Data manipulator network, for N=8. U means use dashed line connection, D means use solid line connection, and H means use dotted line connection. Lower case letters indicate end-around connections.

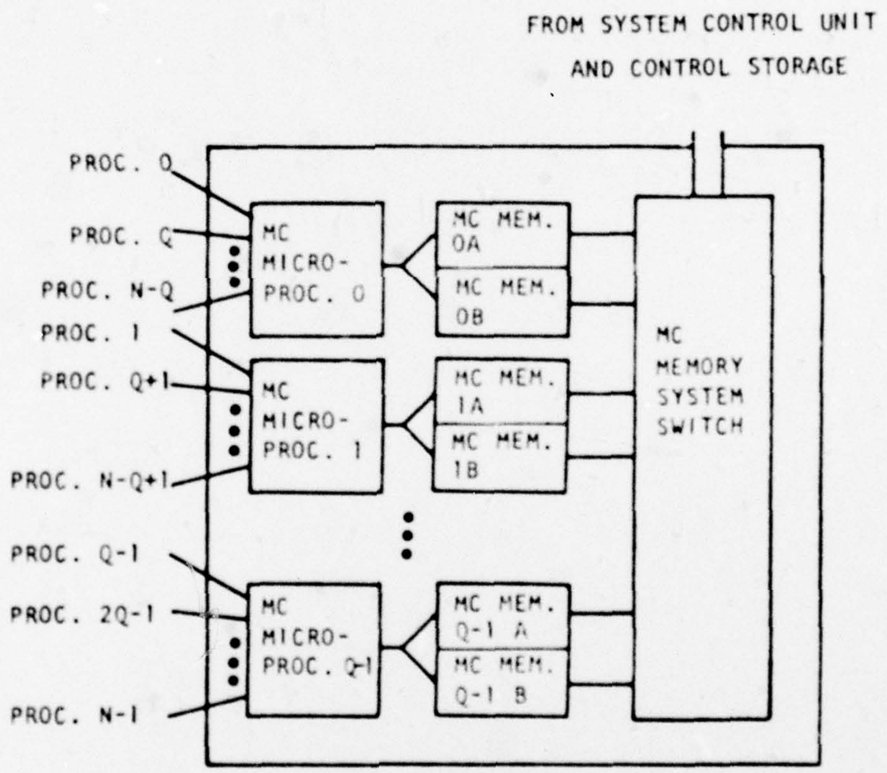


Fig. 8. PASM Micro Controllers.

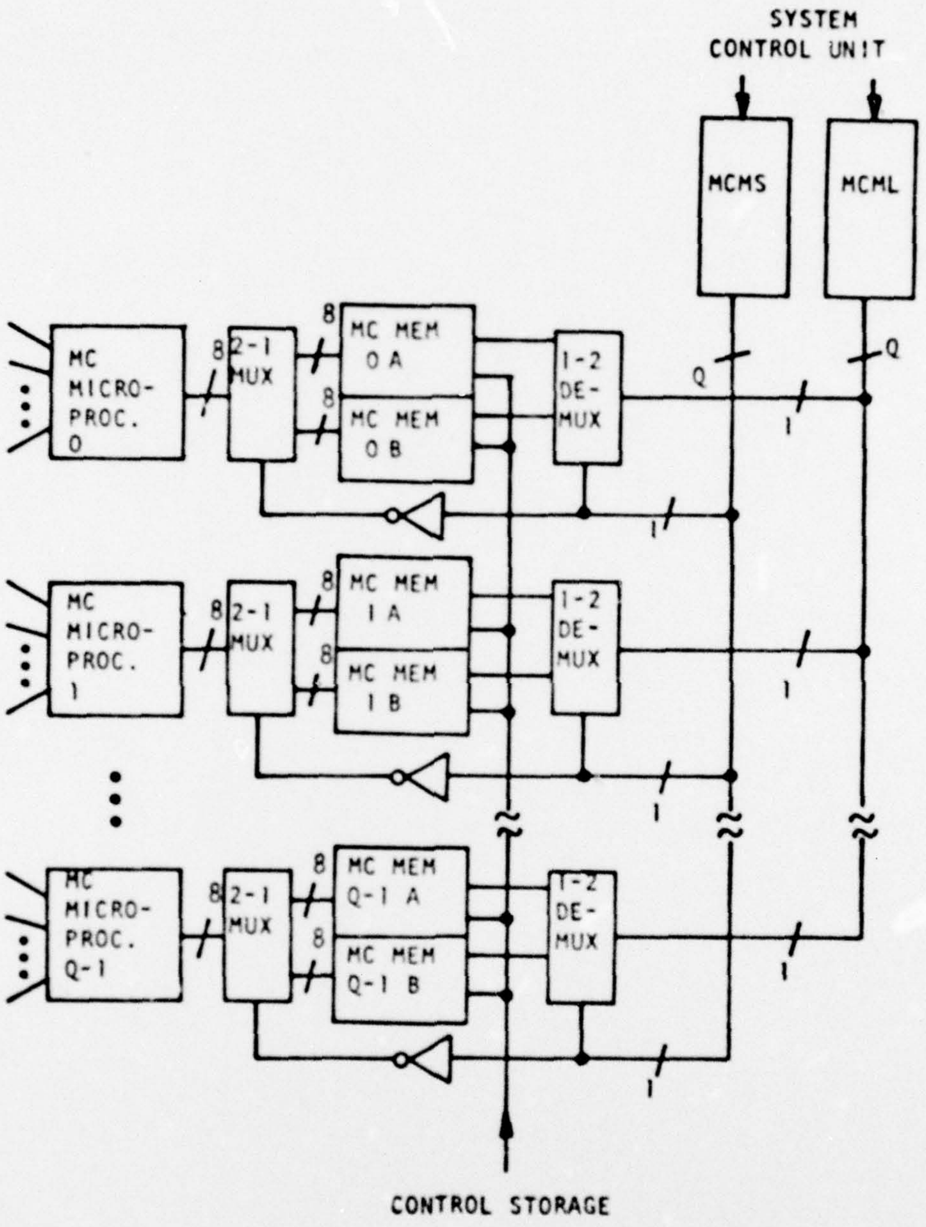


Fig. 9. Communications between the Micro Controllers and the System Control Unit.

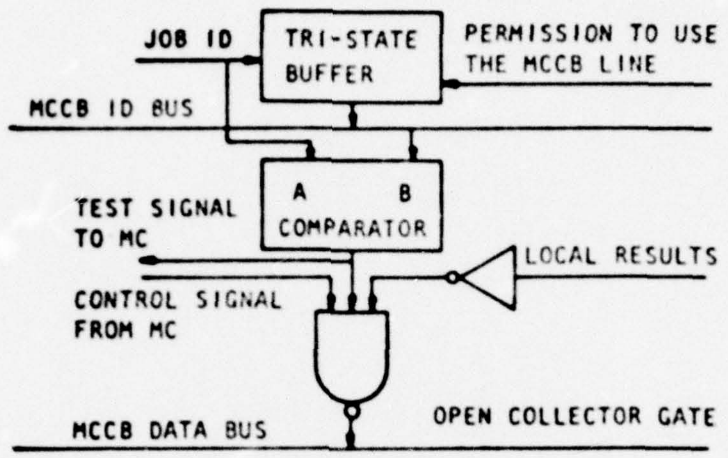


Fig. 10. Micro Controller Communications Bus (MCCB).

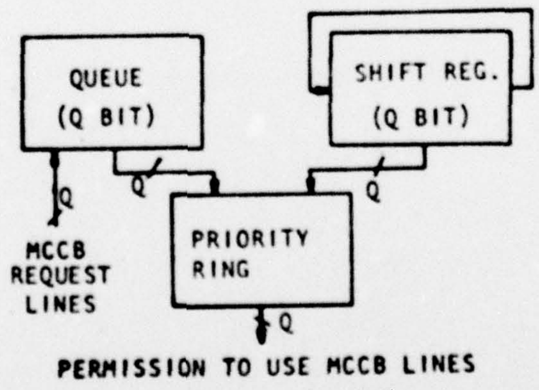
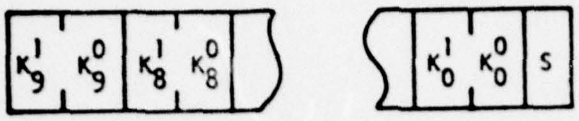


Fig. 11. MCCB controller.



MASK	0	1	X
K^1	0	1	0
K^0	0	1	1

S = 0 POSITIVE MASK

S = 1 NEGATIVE MASK

Fig. 12. PE address mask binary encoding, for N=1024.

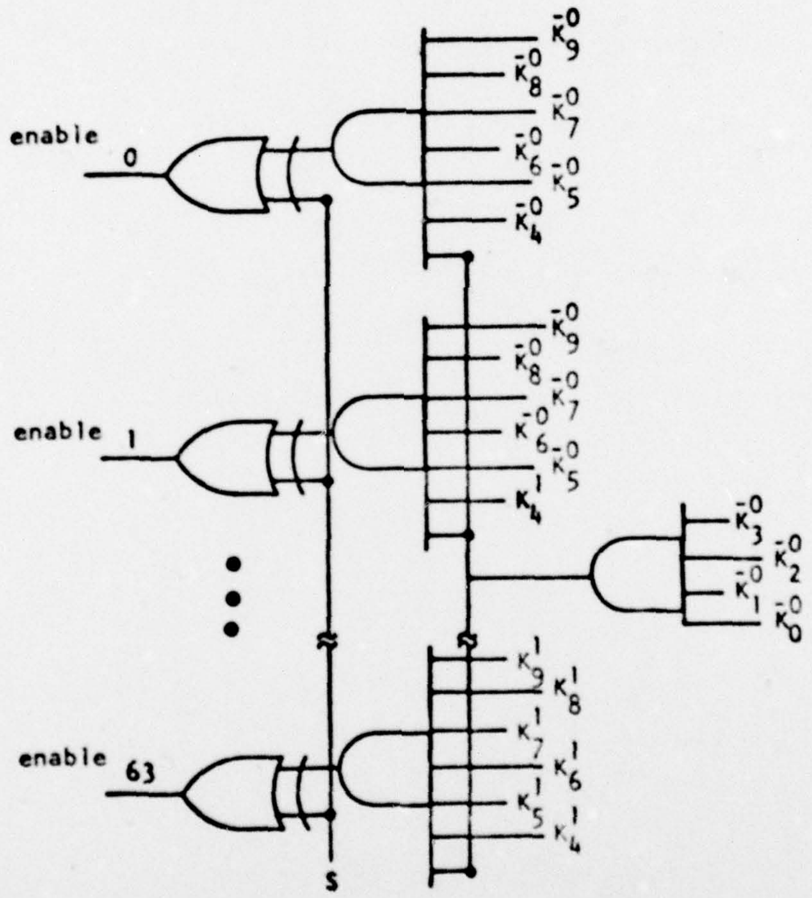

enable 0
enable 1
enable 63
S
$\bar{K}_9^0$ $\bar{K}_8^0$ $\bar{K}_7^0$ $\bar{K}_6^0$ $\bar{K}_5^0$ $\bar{K}_4^0$
$\bar{K}_9^0$ $\bar{K}_8^0$ $\bar{K}_7^0$ $\bar{K}_6^0$ $\bar{K}_5^0$ K_4^1
$\bar{K}_3^0$ $\bar{K}_2^0$ $\bar{K}_1^0$ $\bar{K}_0^0$
K_9^1 K_8^1 K_7^1 K_6^1 K_5^1 K_4^1

Fig. 13. PE address Mask Decoder for the Micro Controller whose physical address is 0.

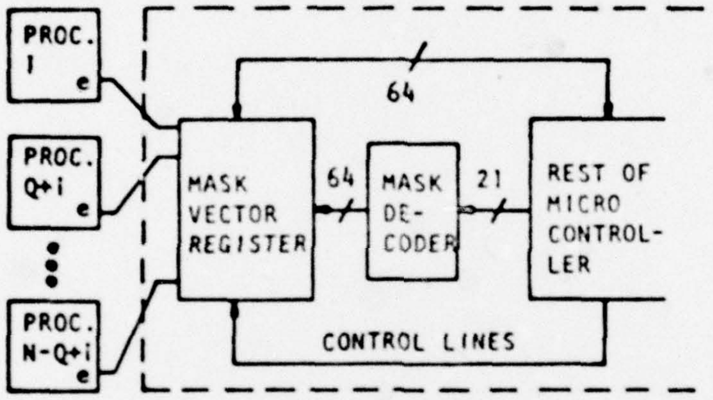


Fig. 14. Masking portion of Micro Controller i. "e" is PCU processor enable bit.

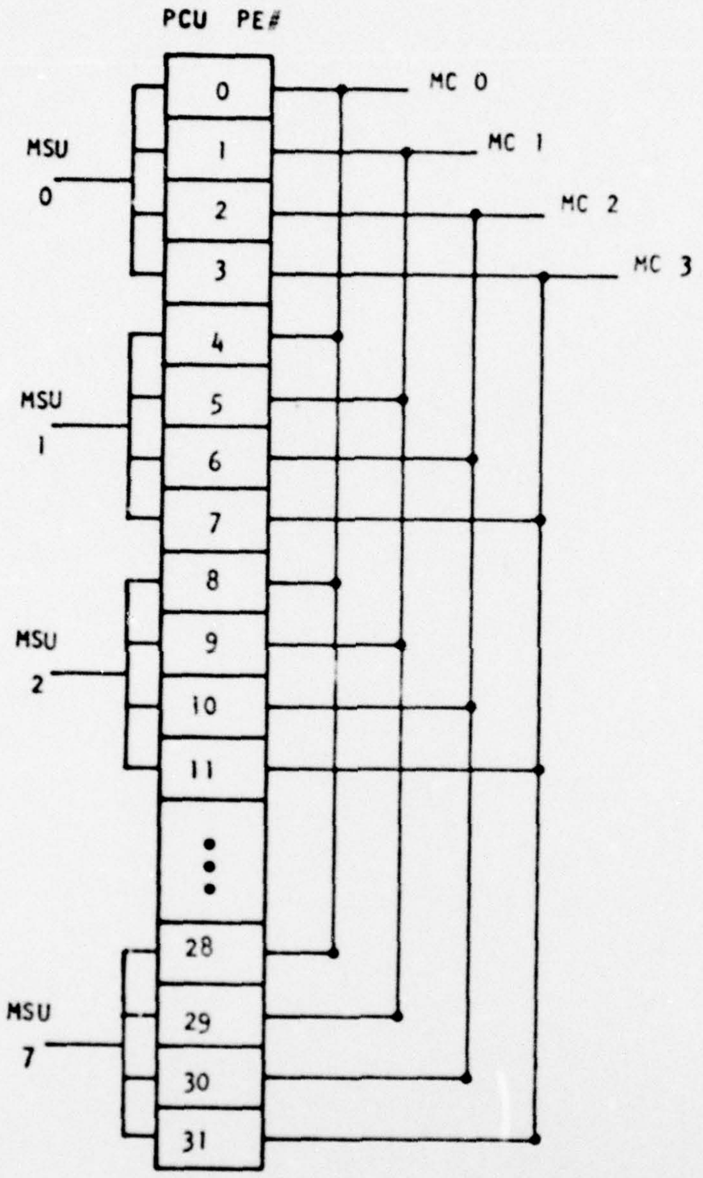


Fig. 15. Organization of the Memory Storage System for N=32 and Q=4, where "MSU" is Memory Storage unit.

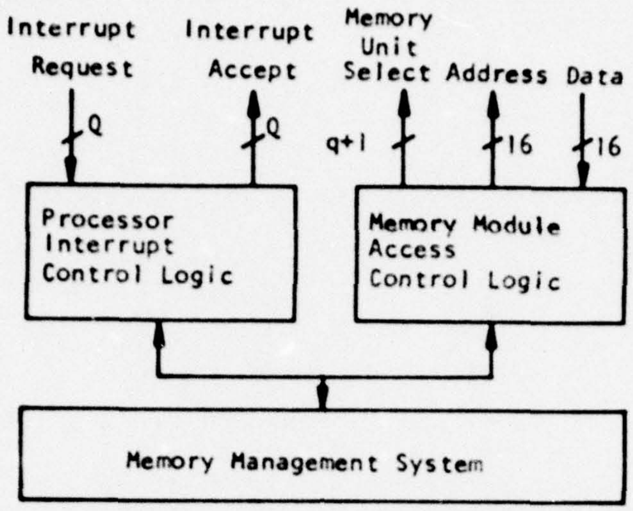


Fig. 16. Hardware scheme for dynamically altering the loading sequence of the memory modules.

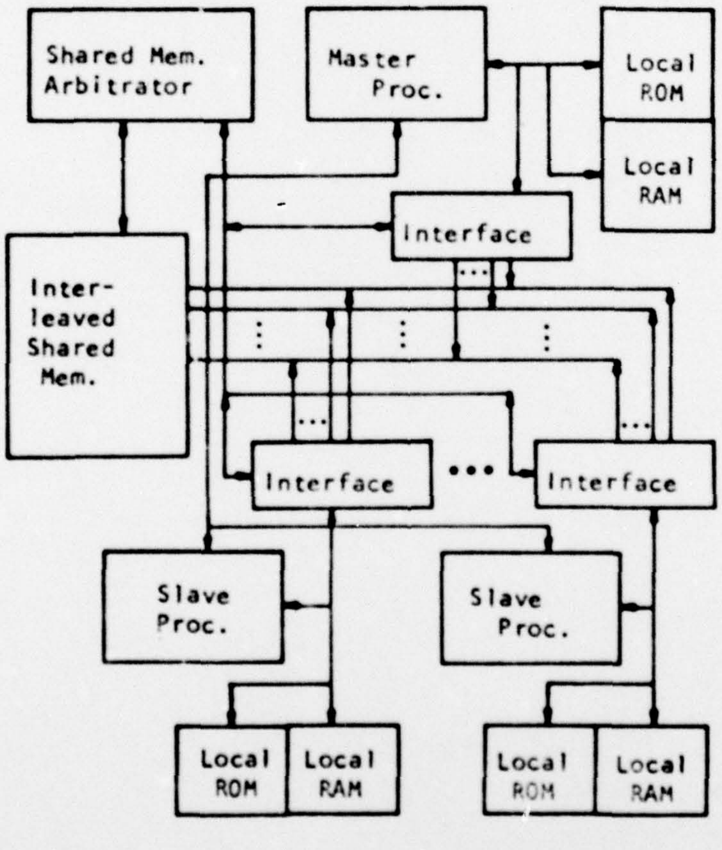


Fig. 17. Distributed Memory Management System.

```
PROCEDURE picture

    /* define pixin and pixout to be 512x512
       arrays of unsigned eight bit integers */

    UNSIGNED BYTE pixin[512][512], pixout[512][512];

    /* define hist to be a 256 word array of integers */

    INTEGER hist[256];

    /* define x and y to be index sets */

    INDEX x, y;

    /* declare pixin to be loaded by input data
       and pixout and hist to be unloaded as
       output data */

   DATA INPUT pixin;
   DATA OUTPUT pixout, hist;

    /* define the sets of indices which x and y
       represent, i.e., x and y represent the integers
       between 1 and 510 inclusive */

    x = y = {1 + 510};

    /* compute average of each point and its
       eight nearest neighbors (simultaneously if possible) */

    pixout[x][y] = (pixin[x-1][y-1]+pixin[x-1][y]+pixin[x-1][y+1]+
                    pixin[x][y-1]+pixin[x][y]+pixin[x][y+1]+
                    pixin[x+1][y-1]+pixin[x+1][y]+pixin[x+1][y+1])/9;

    /* initialize each bin to zero */

    hist[0+ 255] = 0;

    /* compute histogram */

    hist[pixout[x][y]] = hist[pixout[x][y]] + 1;

END picture
```

Fig. 18. High level language algorithm for smoothing and computing histogram. Keywords are in upper case for each of identification, however in actual use they need not be.

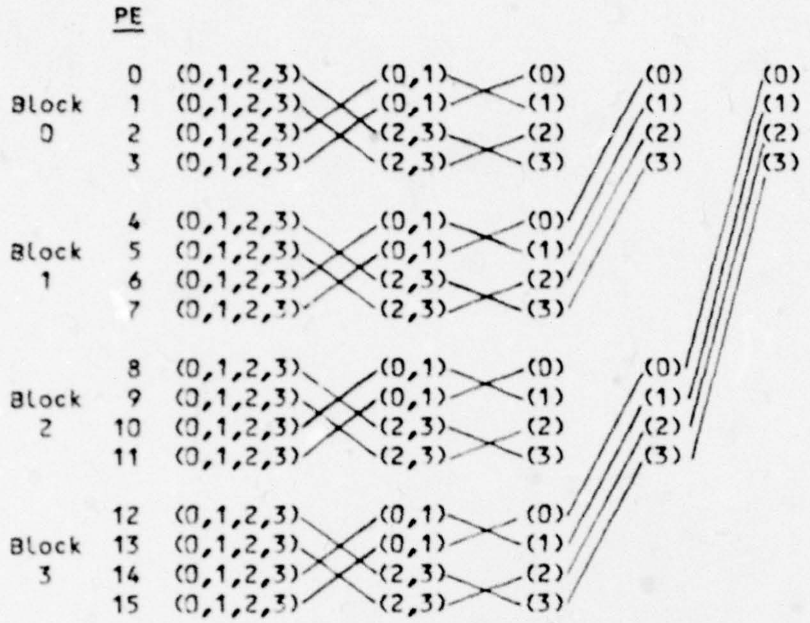


Fig. 19. Histogram calculation for N = 16 PEs, B = 4 bins. (w,...,z) denotes that bins w through z of the partial histogram are in the PE.

```
/*algorithm to combine "local" histograms. $b'=\log_2 B-1$; $n=\log_2 N$;

  keep is the index of the first bin which to be kept.
  send is the index of the first bin which is to be
      sent to another PE */

keep ← 0;
/*form histogram for each group of B PEs */
for i ← 0 to b' do
        /*group of PEs with higher addresses prepares to send first half
          of remaining bins to group of PEs with lower addresses */
        MASK $[X^{n-(b'-i)-1}1X^{b'-i}]$
                send ← keep;
                keep ← send + $2^{b'-i}$;
                SET ICN to PE-$2^{b'-i}$;
        /*group of PEs with lower addresses prepares to send second half of
          remaining bins to group of PEs with higher addresses. */
        MASK $[X^{n-(b'-i)-1}0X^{b'-i}]$
                send ← keep + $2^{b'-i}$;
                SET ICN to PE + $2^{b'-i}$;
        /*transfer $2^{b'-i}$ bins, add received data to kept data */
        MASK $[X^n]$
                DTRin ← hist[send ← send + $2^{b'-i}$-1];
                TRANSFER;
                hist[keep ← keep + $2^{b'-i}$-1] ← hist[keep ← keep +
                        $2^{b'-i}$-1] + DTRout;
/*Combine N/B partial histograms to form histogram of entire image. */
for i ← 0 to $\log_2 N/B-1$
        MASK $[X^{n-b'-i}1X^{b'+1+i}]$
                SET ICN to PE + $2^{b'+1+i}$;
                DTRin ← hist[keep];
                TRANSFER;
        MASK $[X^{n-b'-i}0X^{b'+1+i}]$
                hist[keep] ← hist[keep] + DTRout
```

Fig. 20. Algorithm to perform the B bin histogram calculation for image spread out over N PEs.

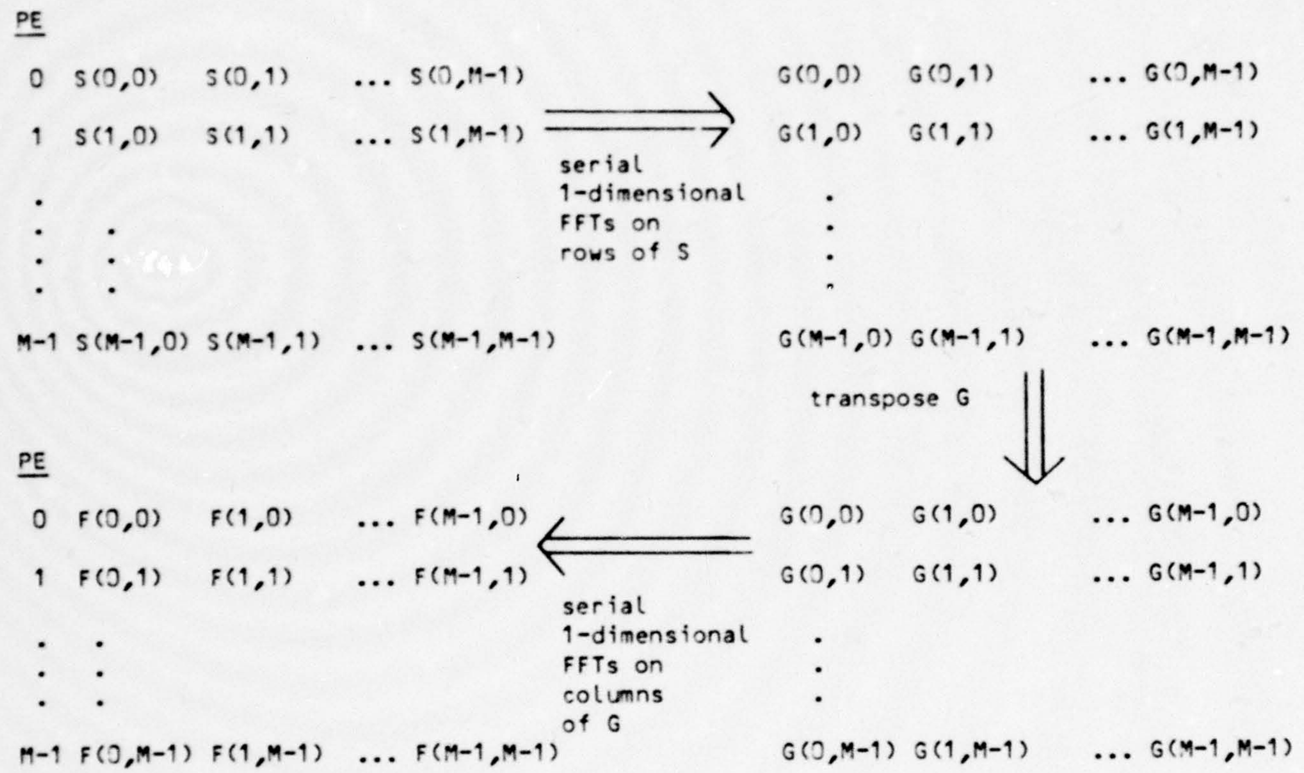

PE
0 S(0,0) S(0,1) ... S(0,M-1)
1 S(1,0) S(1,1) ... S(1,M-1)
M-1 S(M-1,0) S(M-1,1) ... S(M-1,M-1)
serial 1-dimensional FFTs on rows of S
G(0,0) G(0,1) ... G(0,M-1)
G(1,0) G(1,1) ... G(1,M-1)
G(M-1,0) G(M-1,1) ... G(M-1,M-1)
transpose G
G(0,0) G(1,0) ... G(M-1,0)
G(0,1) G(1,1) ... G(M-1,1)
G(0,M-1) G(1,M-1) ... G(M-1,M-1)
serial 1-dimensional FFTs on columns of G
PE
0 F(0,0) F(1,0) ... F(M-1,0)
1 F(0,1) F(1,1) ... F(M-1,1)
M-1 F(0,M-1) F(1,M-1) ... F(M-1,M-1)

Fig. 21. Computation of two-dimensional DFT of M by M array S using M PEs.

```
/* transpose M by M array G, placing result in $G^T$.  PE k receives G(j,k)
   from PE j as follows: At step i, G(j,k)'s for which (k-j) mod M = i are
   transferred from PE j to PE k */

/* In each PE, G denotes the address of the first element of the G vector
   that is stored in that PE; $G^T$ denotes the address of the first element of
   the G transpose vector in each PE. */

for i ← 1 to M-1 do
    /* in PE j, send G(j,k), k = j + i */
    DTRin ← G + b * (ADDRESS + i) mod M;
    SET ICN to PE + i;
    TRANSFER;
    /* in PE k, received G(j,k), j = (k-i) mod M */
    $G^T$ + b * (ADDRESS - i) mod M ← DTRout;
/* copy diagonal from G to $G^T$ */
$G^T$ + b * ADDRESS ← G + b * ADDRESS
```

Fig. 22. Algorithm to transpose an M by M array using M PEs, where initially PE j holds row j of the array.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PASM: A Partitionable Multimicrocomputer SIMD/MIMD System for Image Processing and Pattern Recognition		5. TYPE OF REPORT & PERIOD COVERED Interim March 1978-August 1979
		6. PERFORMING ORG. REPORT NUMBER TR-EE 79-40
7. AUTHOR(s) H. J. Siegel, L. J. Siegel, F. Kemmerer, P.T. Mueller, Jr., H.E. Smalley, Jr., S.D. Smith		8. CONTRACT OR GRANT NUMBER(s) AFOSR-78-3581
9. PERFORMING ORGANIZATION NAME AND ADDRESS School of Electrical Engineering Purdue University West Lafayette, IN 47907		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS United States Air Force, Air Force Office of Scientific Research, Building 410, Bolling AFB Washington, D.C. 20332		12. REPORT DATE August 1979
		13. NUMBER OF PAGES 69
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

FFT, image processing, memory management, MIMD machines, multimicroprocessor systems, multiple-SIMD machines, parallel processing, partitionable computer systems, reconfigurable computer systems, SIMD machines

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Please see reverse side

DD FORM 1 JAN 73 1473

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ABSTRACT

PASM, a large-scale multimicroprocessor system being designed at Purdue University for image processing and pattern recognition, is described. This system can be dynamically reconfigured to operate as one or more independent SIMD and/or MIMD machines. PASM consists of a Parallel Computation Unit, which contains N processors, N memories, and an interconnection network; Q Micro Controllers, each of which controls N/Q processors; N/Q parallel secondary storage devices; a distributed Memory Management System; and a System Control Unit, to coordinate the other system components. Possible values for N and Q are 1024 and 16, respectively. The control schemes, interprocessor communications, and memory management in PASM are explored. Examples of how PASM can be used to perform image processing tasks are given.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)